浙江大学计算机学院

硕士学位论文

基于微内核的调度技术研究

姓名：何家俊

申请学位级别：硕士

专业：计算机系统结构

指导教师：陈文智

20100101

摘要

21 世纪，计算机技术发展越来越快，对操作系统的研究也越来越深入，微内核技术作为一个重要的研究方向，已受到研究人员越来越多的重视，成为了一个研究热点。

本文首先介绍了微内核的相关概念，阐述了微内核的发展历史及技术结构，重点介绍了 L4 微内核的设计框架和技术特点，并深入阐述了它的三大组成部分，线程管理，地址空间和 IPC 机制。在以上研究的基础上，本文研究了微内核的调度策略，并分析了其中的不足之处。调度策略在操作系统中起着重要作用，为用户提供丰富的实时调度策略是一个完善系统应该做到的。而微内核由于追求最小化的设计原则，在内核里只实现了最基本的调度策略，用户需要在用户层实现自身的调度策略。另一方面，用户只能利用微内核提供的接口，获得调度信息，但内核提供的调用接口是有限，并且频繁调用系统调用，对系统的性能会造成比较大的影响。

在对 L4 微内核技术进行深入研究，并分析其调度技术上存在不足的基础上，本文提出了在用户层实现调度器的框架，并实现了优化，本文的主要设计和实现工作为以下三点：

第一，本文设计和实现了一种基于 L4 微内核上的两层调度框架，在用户层实现调度器，通过该框架，用户层调度器可以根据具体调度算法的需要，控制与 L4 微内核和应用线程的通信，获得内核系统资源信息，和应用线程调度信息，实现线程调度策略。该结构适合于用户层调度器获得分布的，多样化的，不同的调度信息。

第二，本文根据设计的框架，在用户层实现了两个调度器，一个是以 EDF 算法为基础的调度器，完全取代内核调度器，在用户层实现调度功能。另一个是辅助调度器，对内核的调度算法进行改进利用，主要调度功能的实现还是在内核中完成。

第三，本文改进了微内核一个系统调用接口，用户层调度器要经过这个调用接口进入内核修改线程调度信息，通过改进该接口，使得调度器通过这个接口进入内核的性能得到提高。

关键词： 微内核，调度器，L4

Abstract

The development of computer technology is becoming faster and faster in the 21st century. Microkernel technology, as an important research direction of operating system, is getting more and more concern by the researchers.

This paper introduces the concepts of microkernel, describes the history of microkernel-based operating systems, the structure and technology. We focus on the design of the L4 microkernel framework and technical characteristics, and go into the details of its three major components, thread management, address space management and IPC mechanisms. Based on the above study, this paper studies the microkernel's scheduling strategy and analyzes its deficiencies. Scheduling strategy plays an important role in the operating system to provide users with a wealth of real-time scheduling strategy. However, in order to keep the size small enough, microkernel implemented only the most basic scheduling policies. Users need to implement its scheduling strategy in the user level to achieve their goals. On the other hand, users can only use limited interface, which is provided by the microkernel, to access scheduling information. But frequently invoking these calls will affect the performance of the system.

Based on the above research, this paper has proposed a solution. The main work of this paper includes the following points:

First, this paper designs and implements a two-layer scheduling framework, which is constructed on L4 microkernel. Through this framework, user-level scheduler can control the communication with the kernel and application threads. Based on its specific scheduling algorithm policy, scheduler achieves kernel system information, and application's scheduling information. This framework is suitable for user-level scheduler to obtain the distribution, diversity, and different scheduling information.

Secondly, this paper implements two simple schedulers. One is an EDF-based scheduler, which completely replaces the kernel scheduler. The other is an assistant scheduler. It helps to improve the kernel's scheduling algorithm. The main process of

the implementation is still completed in the kernel.

Third, this paper improves a system call interface of the microkernel. This improvement helps to reduce the time when user-level scheduler goes into the kernel through this interface.

Keywords: Microkernel, Scheduler, L4

图目录

图 2.1 L4 微内核架构图 9
图 2.2 Exokernel 微内核架构图 10
图 2.3 线程切换流程图 12
图 2.4 内核地址空间划分 15
图 3.1 微内核线程调度策略 23
图 3.2 总体架构图 26
图 3.3 记录控制架构图 28
图 3.4 完全调度器流程图 31
图 3.5 辅助调度器流程图 32
图 3.6 schedule 系统调用二进制接口 36
图 3.7 新通用编程接口 37
图 4.1 地址空间映射示意图 41
图 4.2 内存分配示意图 42
图 4.3 查找数据控制器流程图 45
图 4.4 数据控制器结构图 46
图 4.5 记录内存定位示意图 47
图 4.6 记录数据流程图 49

表目录

表 4.1 微内核接口函数 52
表 4.2 系统环境 56
表 4.3 原系统调用测试时间 57
表 4.4 新系统调用测试时间 57

第1章 绪论

1.1 课题背景

早期的操作系统大多数是宏操作系统，他们把文件系统，网络，设备驱动程序，内存管理，调度策略等等都在内核中实现。现在应用比较广泛的操作系统，例如 Windows，Linux 等，也都是宏操作系统[1]。这些操作系统内核中，有几百万行代码是很常见的事情，各个模块的耦合度，关联度也是比较大，某个地方出现了错误，有可能使得整个系统出现崩溃。

因此，微内核操作系统出现了，它与传统内核的设计理念相反，它的设计是遵循着尽可能使得内核更小，实现更简单的原则。微内核的基本实现方法是把大部分的功能从内核中移到用户层，只保留不可缺少的核心部分，使得内核的复杂度大为降低，内核代码大为缩小[2]。微内核概念刚提出时，在计算机界掀起了一股热潮，很多计算机设计人员和厂家都尝试设计和生产微内核操作系统。微内核的优点是明显的，微内核支持模块化，扩展更加容易；微内核由于代码量小，更容易维护和更新；微内核中各模块的耦合性大为降低，一个模块出现问题不会导致整个系统的崩溃，等等。

但是，早期对微内核的探索和研究大多以失败告终，究其原因，主要是因为性能的问题，第一代微内核以 Mach OS 为代表，它们的性能远远不如宏操作系统，因此，微内核的热潮很快冷却下来。

直到二十世纪 90 年代，微内核开发人员意识到 Mach OS 之所以性能不理想，是由于设计上的缺陷，而不是内在的因素。Mach OS 为了支持一些现在看起来不是很有用的概念，在线程间通讯方面增加了大量的额外系统开销，导致了它的性能大大降低[3]。因此，他们开始重新审视微内核的设计。以 L4 为代表的第二代微内核出现，标志着微内核技术取得了巨大的发展[4]。L4 是一个微内核家族的统称，是最初由 Jochen Liedtke 教授设计的微内核构架的操作系统内核，L4 这个微内核系统由于其出色的性能和很小的体积而开始被计算机工业所认知。相比于第一代

微内核，L4的几个核心性能有了巨大的提高。

当前，基于L4第二代微内核设计思路，实现的两个主要版本是L4Ka::Pistachio和Fiasco。Fiasco是由新南威尔士大学的L4研究组开发实现的，Fiasco是对最初的L4版本的开发，Fiasco是用C++语言写出来的，是为了代码的可读性，以及平台的可移植性。Fiasco包含了对硬实时的支持，并且它的内核允许在任何时间被中断[5]。L4Ka::Pistachio是由Karlsruhe大学的系统结构小组（Systems Architecture Group）开发的，该微内核主要着眼于高性能以及各种平台的支持，当前L4Ka::Pistachio微内核能够支持在大多数硬件上使用[6]。本文的实验也是在L4Ka::Pistachio上设计和实现的。

1.2 研究内容

1.2.1 研究对象

在L4微内核中，只提供了三个最基本的功能，即地址空间管理，线程管理，与IPC通信。其他如设备驱动，文件系统，网络协议以及其他各种服务都是建立在这个内核之外的一些用户态线程。如此一来，一方面系统的扩展性，灵活性和可移植性大大增强。另一方面，实现了内核态与用户态的强隔离，一个运行在用户态的服务的出错不会破坏到内核的运行，从而保证整个系统的连续性。

微内核只提供最简单的功能，因此，一般在内核中不会提供比较丰富的调度策略。L4也符合这个设计原则，在微内核中只提供了一个简单的调度策略，基于优先级的轮转调度，如果用户需要开发比较复杂的调度算法，需要在用户层里实现，在L4Ka::Pistachio中，内核提供了一些系统调用接口供用户使用，可以进行线程切换，改变线程优先级，时间片长度等状态[6]。

1.2.2 提出问题

一般在微内核里实现的调度策略能够比较容易获得系统信息，和线程调度信息。但是在内核里，通常只能实现有限的调度机制，并且，调度机制与内核其他部分也有相关联的，如果在内核中改变调度机制，可能会出现很多复杂的情况。

因此，一般微内核中只实现简单，唯一的调度策略。

L4 微内核作为第二代微内核的典型代表，它的设计是遵循着尽可能使得内核更小，实现更简单的原则。内核里面的调度机制的设计也是遵循这个原则，L4 实现的调度策略是基于优先级的时间片轮转调度，该策略实现较为简单，也基本满足系统实时性的要求。但是，对于一些软实时或非实时的低优先级线程，可能会出现饿死的情况[7]。

另外，如果用户想根据不同的情况，用不同的调度策略，特别是对一些虚拟机上的应用，例如在 KVM 虚拟机上运行的 3D 图形应用，对系统资源的调度策略要求比较高[8]，但是，原来的微内核是没有办法提供的。因此，需要用户在用户层实现具体的调度策略，但是，这又会产生另外一个问题，在内核里实现调度策略的时候，可以比较容易地获得系统资源信息，或者线程调度信息。在用户层，只能获得有限的内核信息，因为在内核可以访问内核内存，但在用户层是禁止访问的，只用通过调用内核提供的接口，获得内核信息。而要实现比较丰富的调度策略，用户层调度器可能会根据调度策略，需要得到额外的调度信息，但是，现在的内核设计是不提供的。

1.3 研究目标

调度策略在操作系统中起着重要作用，为用户提供丰富的实时调度策略是一个完善系统应该做到的。而微内核由于追求最小化的设计原则，在内核里只实现了最基本的调度策略，用户需要在用户层实现自身的调度策略。但另一方面，在用户层是不能访问内核内存的，用户只能利用微内核提供的接口，获得调度信息，但内核提供的系统调用接口是有限，并且频繁调用系统调用，对系统的性能会造成比较大的影响。

因此，本文在研究了微内核的技术现状，以及当前实现方法的不足的基础上，设计和实现了一种基于 L4 微内核上的两层调度模型，在用户层实现调度器，用户可以根据自身需要，获得内核系统信息，和应用线程调度信息，实现线程调度策略或系统资源调度策略。该结构适合于用户层调度器获得分布的，多样化的，

不同的调度信息。本文在内核中设置记录点，当事件触发后调用记录函数，用户层调度器可以根据具体调度算法需要，记录事件点的相关调度信息，以自定义格式存入内存中，以地址映射的方式提供给用户层调度器，由调度器解读信息。

另外，本文根据设计的框架，在用户层实现了两个简单的调度器，一个是以EDF（Earliest Deadline First）算法为基础的调度器，完全取代内核调度器，在用户层实现调度功能。另一个是实现了辅助调度器，对内核的调度算法进行改进利用，主要调度功能的实现还是在内核中完成。说明了能根据设计的框架，在用户层实现调度器。

再者，本文改进了一个系统调用接口，提高了用户层调度器通过这个接口进入内核的性能。

1.4 本文组织结构

本文一共有五章，组织结构如下：

第一章：绪论

介绍了本文的课题研究背景以及研究内容，针对研究内容提出了相关问题，并根据存在的问题，提出了研究的目标和解决方法。

第二章：微内核

介绍了微内核的相关概念，阐述了微内核的发展历史及技术结构，简要介绍了第一代和第二代微内核技术的发展，以及当前的代表性微内核。最后，重点介绍了 L4 微内核的设计框架，并深入阐述了它的三大组成部分，线程管理，地址空间和 IPC 机制。

第三章：用户层调度器的设计

本章设计和实现了一种基于 L4 微内核的两层调度模型，在用户态实现调度器，调度器获得内核系统信息，和应用线程调度信息，实现自身的线程调度策略或系统资源调度策略，并且调度器通过记录控制器控制获取哪些数据。另外，本章根据设计的框架，在用户层设计了两个简单的调度器，一个是以 EDF 算法为基础的调度器，完全取代内核调度器，在用户层实现调度功能。另一个是实现了辅

助调度器，对内核的调度算法进行改进利用，主要调度功能的实现还是在内核中完成，证明了能根据设计的框架，在用户层实现调度器。再者，本文改进了一个系统调用接口，提高了用户层调度器通过这个接口进入内核的性能。

第四章：用户层调度器的实现

本章依据第三章设计的调度器模型进行实现，详细介绍了用户层调度器与内核、应用线程的通信机制，信息控制机制，和数据记录过程。并依据这个框架，在用户层实现了两个调度器。最后，介绍了如何实现 schedule 系统调用接口的改进，提高用户层调度器通过这个接口进入内核的性能。

第五章：总结和展望

本章首先对全文进行了总结，回顾了本文的主要研究内容，以及归纳出几个方面的主要工作，并指出下一阶段的研究内容，和深化的方向。

1.5 本章小结

在这一章中，我们介绍了课题研究背景，主要介绍了微内核的兴起，以及当前开发情况等。在对研究对象进行简要分析的基础上，提出了当前存在的问题和不足，并根据提出的问题，给出了具体解决方案和研究目标。

第2章 微内核

2.1 微内核相关内容

2.1.1 微内核概述

微内核的概念最早是由 Richard Rashid 提出来的，当时，他在卡内基梅隆大学带头开发 Mach 操作系统，目标是建立一个基于消息传送机制的微小内核。微内核是操作系统的核心，它包含系统中必须用到的功能，包括线程管理，地址空间控制，线程间通信等[9]，这些是操作系统的核心功能。一些非必要的功能留在用户层实现，这样，微内核就能实现较小的规模。

为了在微内核上实现各种功能，可以在内核提供的基础服务上，在用户层实现各种其他服务，这些是微内核以外部分，例如包括文件系统，网络服务，设备驱动等等。这些是以模块的方式加载进系统，内核与服务以及服务模块之间是用消息传递机制（IPC）来通信的。该传递机制使得各种服务与内核联系在一起，形成一个整体，这些提供服务的模块与微内核就组成了操作系统[10]。所以，各种应用程序可以在这个操作系统之上运行。

直接与硬件打交道的是微内核，它在内核态下工作。操作系统的其它非必要功能在用户层实现，这些部分被分成多个相对独立的进程，每个进程提供一种服务，提供的对象可以是独立的应用程序，也可以是另外一个服务器。这种服务器模型就是把操作系统作为分为许多相互独立的协作进程，它们为用户提供各种形式的服务。而服务器通过系统调用接口与微内核联系。各服务模块之间相互独立，彼此之间通过 IPC 机制通信。微内核对对各种来往消息进行验证，在各大部分之间进行消息传递，并保证它们对硬件的访问[11]。

微内核的设计与传统宏内核相比，一个重要的优点是提高了系统的灵活性以及扩展性。在前面提到微内核只包含操作系统必不可少的几个部分，因此，可以把微内核看作是对传统宏内核共性的总结，把宏内核中必要的部分提取出来，其他可以从必要部分衍生出来的，放到用户层中实现。换句话说，微内核只提供了机制和资源，而具体实现策略留给用户层线程来实现[12]。当需要添加新的软件服

务或者硬件设备时，只需要增加一个服务模块即可，而不需要像宏内核操作系统那样要从新修改内核设计。

2.1.2 第一代微内核

第一代微内以 Mach OS 为代表，Mach 是一个微内核操作系统，它是在卡内基梅隆大学开发出来的，支持操作系统的研究，分布式和并行计算。它是最早实现的微内核操作系统的例子，对以后微内核的研究起着重要作用。

Mach OS 项目实施在 1985 到 1994 年期间，一直到 Mach 3.0 的开发完成才结束。Mach 微内核提供了线程管理、内存控制、线程间通信和 I/O 服务等功能。在 Mach 微内核基础上，建立了一些运行在用户态线程的操作系统模拟程序，用来取代 UNIX 的 BSD 版本，模拟各种不同操作系统的特性。因此，Mach 能够运行不同操作系统上的应用程序[13]。Mach 微内核的设计带来了一个重要的优点，就是极大地提高了系统的兼容性，使得基于微内核的操作系统能够模拟其它操作系统的特性，从而支持许多运行于其它操作系统上的运用程序。

当时，Mach 微内核是基于传统的 UNIX 内核来设计，也是为了取代 UNIX 内核的，Mach 微内核与 UNIX 的主要不同是，取消了在 Unix 中任何东西都是文件的概念，Mach 能处理任何任务，越来越多的代码被移出内核，移到用户层来实现，形成了一个更小的内核。与当时传统的系统不一样，在 Mach 中，一个线程，或者说一个任务，能由一组线程构成。在现代操作系统中，这个看起来很普遍，但在当时，Mach 是第一个系统以这种方式定义进程和线程的。内核的工作从作为一个完整的操作系统，到仅仅维持任务，并安排他们获得硬件资源。

以下几个概念是 Mach 内核设计的要点：任务的概念，任务是一组资源，这些资源能够使线程运行起来；线程的概念，线程是运行在一个处理器上的一个单元的代码；端口的概念，端口定义了一条任务之间进行 IPC 通信的安全管道。消息的概念，消息是通过端口在任务间传输的[13] [14]。

2.1.3 第二代微内核

第一代微内核的提出掀起了一股小旋风，微内核的设计具有很好的灵活性，

扩展性，兼容性等等。但是，第一代微内核的性能与传统宏内核相比却是比较差，究其原因，是由于第一代微内核的设计有一个重要缺点，由于微内核系统用独立的进程来隔离系统服务，这些服务之间的通信模式是采用消息传递的方式，这实际上进行了一次远程系统调用（RPC）。但是在第一代微内核中实现以后，发现性能比较差，低于一般系统调用的性能。因为微内核进行一次这种消息传递，需要进行多个步骤，包括创建及发送消息，进行线程切换，等待对方接收消息等，这些成为了系统消息传输性能的瓶颈。而微内核很多功能都是基于消息传递机制实现的，系统需要经常传送消息，消息传输的性能瓶颈造成了系统整体性能大大降低[13]。

为了解决这个问题，当时提出了两个方法，一种方法是重新扩大微内核，把一些服务和模块放入内核中，通过扩大内核的办法，减少系统在用户层和内核态进行切换的频率，和在用户层服务间进行通信的频率。例如，在微内核概念出现的时候，Windows 把图形系统从内核中拿出来，放到用户层实现。而在 Windows NT 版本中，又把部分图形系统放回到内核中去，结果这部分性能有了很大地提升。

但是通过重新扩大内核的方法，给系统性能带来提高，与微内核的设计理念不符，最后，可能又会回到宏内核的老路上。因此，一些微内核设计人员提出了另外一种解决方案，这种方法的思路与第一种恰恰相反，通过进一步减少内核的大小，以及对消息传递机制进行优化，来实现整个系统性能的提高。

下面将会概要介绍几个典型的第二代微内核。

（1）L4 微内核

L4 微内核是第二代微内核的代表，与 Mach 为代表的第一代微内核相比，L4 在 IPC 性能上有了极大的提高，它的 IPC 比第一代微内核的 IPC 快 20 倍以上，达到甚至超过了传统宏内核的消息通信性能。在实际上，也有了更广阔的应用价值。L4 微内核包括两个抽象，分别是线程，以及保护线程之间互不干扰的地址空间。为了能够操纵这些抽象，L4 还提供了两个机制，进程间消息通信机制（IPC）和地址映射机制[15]。前者使得线程可以通信以及同步，而后者则使得可以递归地

复制地址空间。L4 微内核的结构图如图 2.1 所示，具体的技术细节将在下一节详细阐述。

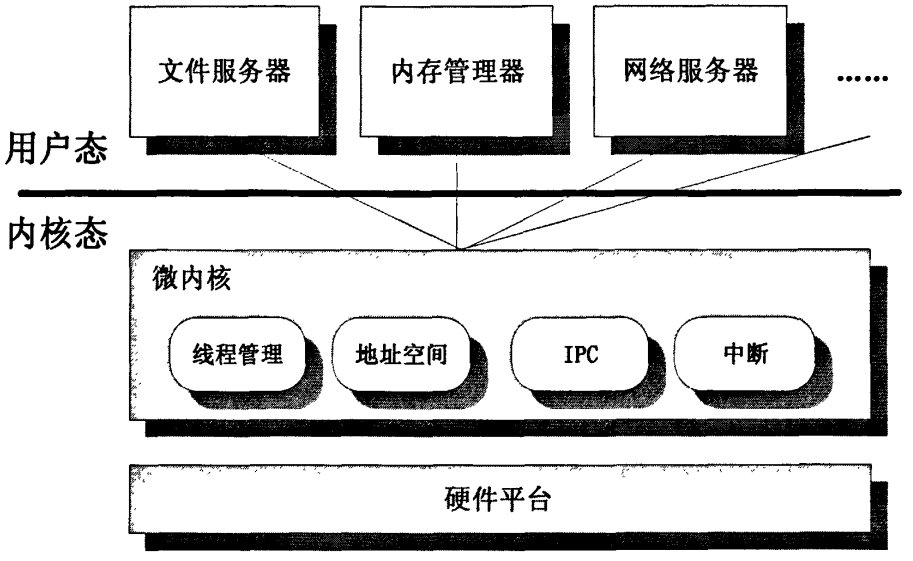

图 2.1 L4 微内核架构图

（2）Exokernel 微内核

另外一个第二代微内核 Exokernel，这个内核是由微内核是由 MIT 的 PDOS 团队（Parallel and Distributed Operating Systems group）开发的。与 L4 微内核相比较，它是一个精简得更加彻底的内核，该内核的思想是，在内核中应该提供尽可能少的系统资源的抽象，而让用户尽可能多地决定这些抽象。以 Exokernels 内核的观点来看，IPC 是一个太高级的抽象，因而不能达到最高的性能。Exokernels 内核是很小的，因为它的功能仅限于保护和复用资源，因此，相比于传统微内核的消息传递机制和宏内核的抽象实现，它的实现大为简化[16]。

Exokernel 的结构如图 2.2 所示，在上层实现的应用程序被称为库操作系统（library operating systems），他们可以要求特定的内存地址，磁盘块等，微内核仅仅确保请求的资源是可以获得的，应用程序可以访问它。这样，程序员可以决定和实现自定义的抽象，并省略掉不必要的，这种方式能够提高程序的性能。

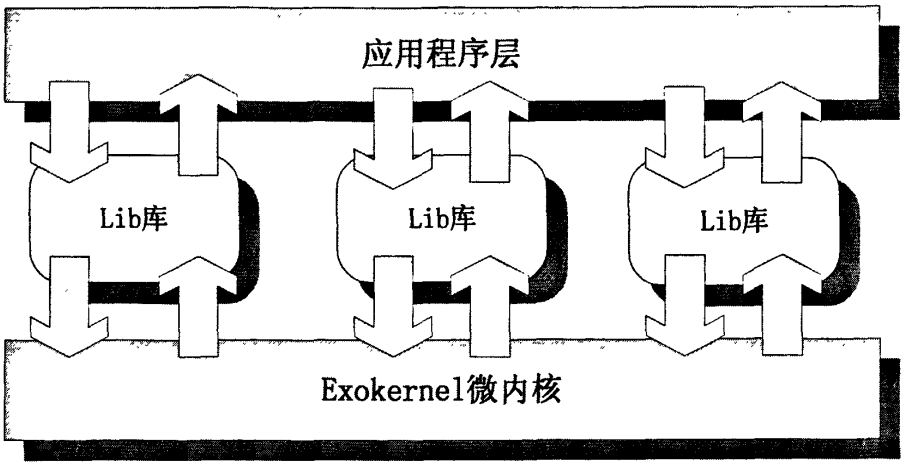

图 2.2 Exokernel 微内核架构图

（3）Rambler 微内核

另外一种基于一个第二代微内核的操作系统是漫游者（Rambler），Rambler 是一种高扩展性高容错性的内核，它与传统操作系统不一样，严格来说，它不像传统的操作系统，它是没有内核的。它通过支持不同地址空间和跨计算机平台的调用，把整个系统联系起来，每个部分间形成一条桥梁通信。

在 Rambler 操作系统中也有一个内核态进程，但和传统操作系统的内核有很大不同，Rambler 的内核进程只是一个运行在内核态的地址空间。在这个内核态地址空间中有什么系统部件，是不可预知的，取决于处理器的体系结构。所以，Rambler 内核进程实际不同于传统操作系统的内核。而传统操作系统内核的主要功能被 Rambler 系统的各个子系统代替了。因而从某种意义上说，Rambler 操作系统可以被认为是一种无核设计[17]。

2.2 L4 微内核技术

2.2.1 线程管理

2.2.1.1 线程概念

线程管理是 L4 微内核的核心之一，线程状态用数据结构 TCB（thread control

block）来表示，该数据结构包含 CPU 状态，当前指令指针，堆栈指针，通用寄存器和特殊寄存器的内容（例如，浮点寄存器和状态寄存器等）。另外，为了能够控制线程切换和调度，该数据结构也存储线程优先级，时间片长度，线程状态（running, ready, blocked, invalid 等）等调度参数。

L4 的线程状态主要分为运行状态，就绪状态，等待状态，挂起状态。运行状态表示处于该状态的线程可以使用 CPU，执行它的指令，在一个时间点，只有一个线程处于该状态[18]。就绪状态是运行状态的前提，线程要转换成运行状态，首先必须处于就绪状态，然后等待当前运行线程时间片到或者被抢占，由调度器决定接下去哪个处于就绪状态的线程取得处理器执行权力。等待状态表示线程不可以继续执行，它需要等待起码一个资源或事件。挂起状态表示处于该状态的线程是被动进入该状态的，它可以被激活，继续执行。

2.2.1.2 线程切换机制

在 L4 操作系统中，切换线程主要包括两个步骤，第一，存储当前运行线程的所有 CPU 状态到它的 TCB；第二，把下一个线程的 CPU 状态从它的 TCB 拿出来，加载到 CPU。

线程 A 切换到线程 B 的过程如图 2.3 所示，当前线程 A 占用 CPU，要进行线程切换，首先必须保存线程 A 状态到线程 A 的 TCB 结构，然后访问线程 B 的 TCB 结构，并加载线程 B 的 IP，SP 指针及其他寄存器状态到 CPU，完成线程切换，该过程主要是在内核态中完成的。

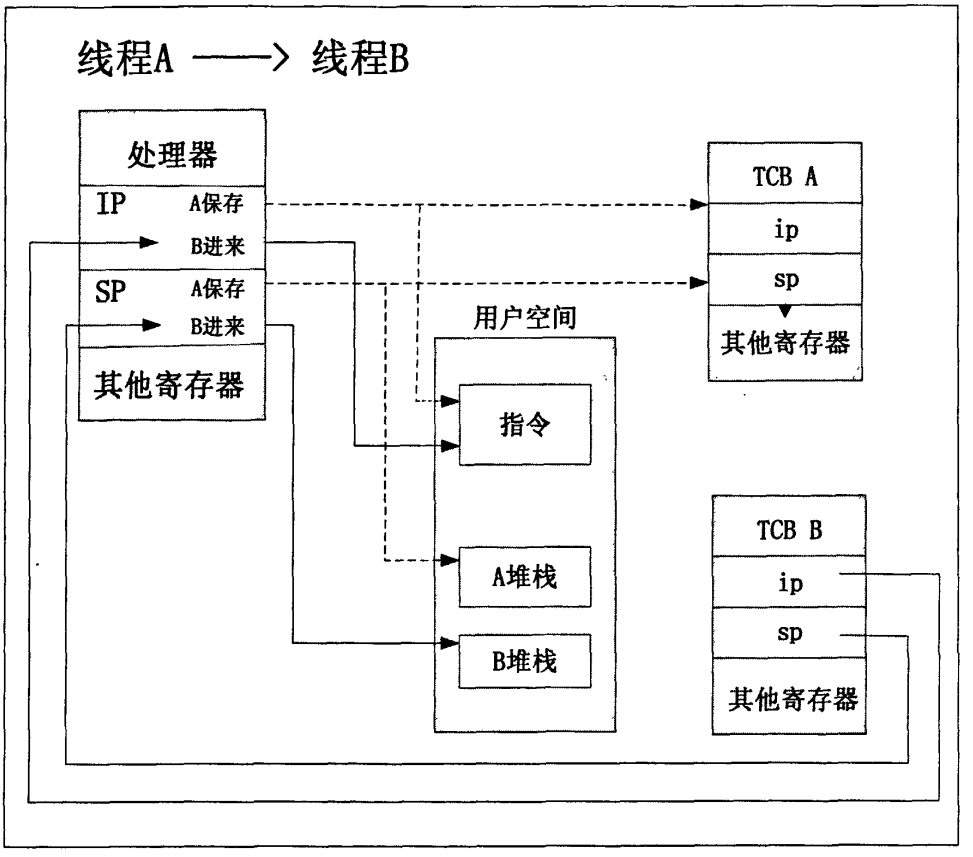

图 2.3 线程切换流程图

线程 TCB 是受保护的，在 L4 中，TCB 是内核对象，在用户态的线程不能修改其他线程的 TCB，甚至是不能修改自己的 TCB[18]。因此，线程切换必须在内核中完成。当前在内核态中运行，线程在用户层的状态必须保存起来，以便从内核态返回时可以恢复。

线程切换需要下列步骤：

- 线程运行在用户态。
- 某个事件导致了线程切换的发生（中断，异常，系统调用）
- 系统进入内核态。
  - 硬件自动存储用户态堆栈指针。
  - 加载内核堆栈指针。

- 存储用户态指令指针和堆栈指针到内核堆栈。
- 硬件基于进入内核的方式（中断，异常，系统调用）加载内核指令指针。

- 内核保存剩余的 CPU 状态到当前线程的 TCB。
- 内核从下一个线程的 TCB 中加载 CPU 状态。

从内核态返回到用户态。

2.2.1.3 线程 TCB

一个线程的 TCB 存储了线程 CPU 的状态（堆栈指针，指令指针，寄存器等），除了状态，线程的管理数据也存在 TCB 中，包括：全局线程 ID（用来匹配哪个 TCB 属于哪个线程）；局部线程 ID（为了在内核态中获得 UTCB）；在切换过程中，线程当前用到的资源；线程状态（ready, waiting, locked running 等）；不同的调度参数，例如线程优先级，总共的时间片长度和剩余的时间片长度，以及该线程被允许运行的最大 CPU 时间（当前剩余的）；双链表结构，链接与该线程状态一样的所有线程（等待状态的线程列表，准备状态的线程列表等）；指向描述线程地址空间数据结构的指针（例如，指向页目录的指针）；硬件地址空间标识符（对于 Intel，就是页目录物理地址的起始地址 cr3）[19]。

线程状态列表看起来是多余的，但在一些情况下，可以加快获得下一个线程。当要决定下一个执行线程时，每个优先级的准备状态列表就可以用到，这样，继任者就能被选出来。另外，状态列表是放在 TCB 中，所以不需要对列表项额外的分配或释放空间。这简化了内核内存的管理。再者，TCB 中保证了列表指针和 TCB 其他参数在同一个内存页里，因此，全部线程相关的数据共享同一个 TLB 入口，避免了额外的 TLB 缺失。

线程对应的 TCB 是根据线程 ID 来查找的，在 L4 内核中，并不是建立一个线程才分配一个 TCB 的，而是在系统开始的时候，就分配虚拟地址空间给所有的 TCB。为了简化 TCB 的管理，所有 TCB 分配在一个连续的虚拟地址空间里，由线程 ID 索引。如果在一个 32 位的系统里，每个 TCB 大小为 2kByte，这样可以

有21位来表示线程号（大概有2M个线程），但会用完4G的地址空间。所以，在内核中，限制了线程的数量为256k（18位），因此，需要512MB虚拟空间[20]。

在L4中，32位的线程ID实际上只用到18位表示，每个线程ID对应一个TCB，其他14位作为版本号，因为考虑到在不同的系统时间段，或许可以重用线程ID，在内核中，版本号是没用的，留给用户层线程或任务来用的。在查找对应TCB过程中，线程版本号是忽略的，线程ID加上TCB列表的起始偏移就找到该ID对应的TCB。

2.2.1.4 线程调度

在内核中，找到下一个要运行的线程可以是很费时的操作，因此这个过程必须是可抢占的，从而保证较小的中断延时。如果调度器要在当前线程（即要被切换的线程）的上下文中运行的话，这个目标会比较难实现，当有中断的时候，被中断的线程的当前状态要保存下来，内核中的中断处理器会仍然运行在当前线程的堆栈上，有可能会再去调用调度器。当当前线程被重新调度时，它会继续它之前被抢占的调度活动，虽然该线程本身本应该是继续运行的[6]。此外，中断处理器以及嵌套的调度器都会在线程的内核栈上消耗空间，而这些都在TCB里，会非常受限。

为了解决所有这些问题，L4使用了一个专门的线程来进行调度选择。这个线程在被抢占之后不会恢复，而是完全重新开始，丢弃原来所有的状态（调度相关的数据结构在调度器被抢占之后有可能发生变化，所以一个全新的开始是比较好的）[21][19]。这样同时也避免了要有一个非常大的内核栈的需要，因为除了单个的中断处理器之后，不会再有任何的东西嵌套在调度器之上。

在这个模型之下，通过调度器调度的过程，包含两个线程切换：第一个是从当前线程到调度器线程，第二个是从调度器线程到下一个运行的线程。然而，由于调度器永远不会运行在用户态下，不会有一个对应的地址空间，因此切换的代价很低（没有地址空间切换，没有TLB刷新）。由于它永远不会被用户态下的东西所引用，故它甚至可以不需要线程id，从而可以自由地放置它的堆栈，并且它

将不会被包含在任何的 TCB 中[22]。

在 L4 微内核中，每个线程都有一个优先级，优先级的范围从 0 到 255，当要调度下一个运行线程时，微内核会采用基于优先级的轮转调度机制，该机制将会在下一章中具体介绍。

2.2.2 地址空间管理

2.2.2.1 地址空间划分

在 L4 微内核 32 位（4G）虚拟地址空间中，分为用户地址空间和内核地址空间，内核地址空间存储内核代码和数据。3G 给用户层，1G 给内核。每个地址空间中，高位的 1GB（0xC0000000...0xFFFFFFFF）留给内核使用，对于所有地址空间，内核的 1GB 空间都是同步的，处于不同地址空间的线程，看到的内容是一样的。内核的 1GB 空间分为：每个地址空间自己使用的 128MB，TCB 项 512MB，内核代码和数据 128MB，最后的 256MB 作为直接映射到物理内存首 256MB 的窗口，以便有效操作（例如，页目录等）[23][19]。L4 微内核地址空间的具体划分如图 2.4 所示。

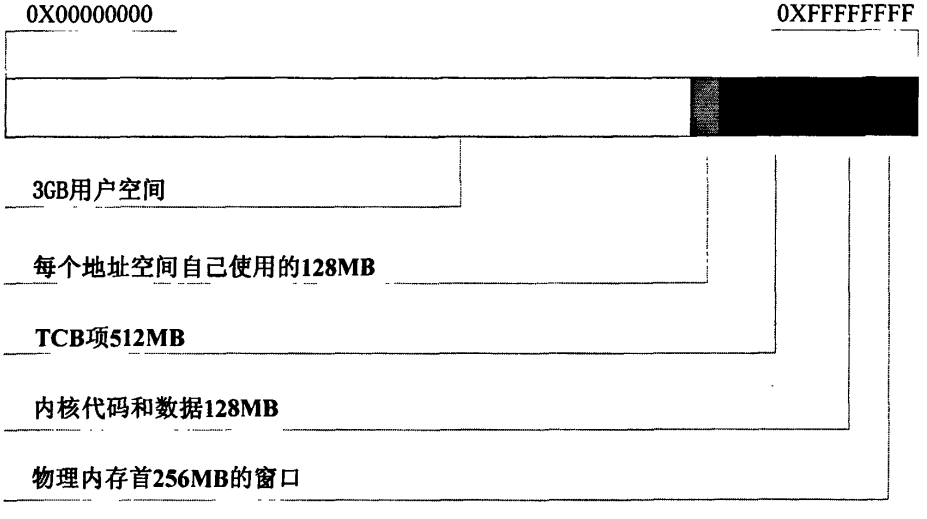

图 2.4 内核地址空间划分

2.2.2.2 地址空间同步

内核地址空间中的大部分东西都是静态的，包括代码，数据，映射到物理地址的窗口，或者本来就不需要同步的（每个地址空间的独立区域），这两部分都不需要关心同步的问题[18] [19]。但是，对于 TCB 列表，它是动态的，必须在每个地址空间中保持一致。

如果地址空间的 TCB 不一致，假如有这种情况，线程 A 创建了线程 N，那么，在 A 地址空间中，内核创建了新的 TCB 映射（线程 N 的 TCB 映射到一页，还没有被用），并继续执行 A。当 A 时间片用完后，内核决定执行在另一个地址空间的线程 B，线程 B 想要与 N 通信，检查 N 是否存在，但通过查找 B 地址空间的 TCB 列表，发现 N 线程并不存在。更糟糕的情况是，B 也创建了另一个线程 N，现在，内核出现了不一致的情况，两个 TCB 有同一个线程号，全局 ID 就不是唯一的了。

为了处理这个问题，所有的 TCB 项映射都放在一个管理页表里，当 TCB 列表发生页缺失时，管理页表的映射被读取，并复制给缺失的页表，这样，发生缺失的线程就可以读取 TCB 数据，如果是存在的线程，则映射到实际的页，就可以读到正确的 TCB，如果是映射到 0-page 的，就知道不是合法的读取 TCB 操作。如果在写时发生缺页，那么，只有在管理页表中，不是映射到 0-page 的，才会把映射复制给缺失页表[23] [19]。在上面的例子中，新的线程被分配一个新的物理页，并把这个映射放进管理页表，这步完成后，新的映射关系才会复制到发生缺失的地址空间的页表。

这样不需要每次更新所有地址空间的页表，就能保证一致性。但取消或者修改映射可能会出现问题，有以下两个原因保证这不会出问题[6]：

第一，内核不会释放分配给 TCB 的页，所以，不会发生取消映射的情况。

第二，在 Intel 结构中，采用了两级页表，在所有地址空间中，保存的是最高层的页目录，而 TCB 的页表都是一样的，所以，改变某一个页表项映射不需要修改页目录。但如果页目录项发生改变，则需要更新每个地址空间。

2.2.2.3 处理器本地数据

不同的处理器有自己专门的数据（例如指向准备状态线程队列的指针），从而避免多个处理器出现数据冲突。CPU 本地数据可以用两种方法实现，一种是对每一个结构都用一个列表，根据 CPU 号来索引，找到数据；另一种根据不同的 CPU，把他们的本地数据固定在不同的物理页上。后一种方法更简洁，代码不会混乱，不需要为每个数据构造列表，而是数据结构都在相同的虚拟地址里，只是根据处理器的不同，映射到不同的页里。所以，在 L4 里，采用的是后一种方法[23]。这种方法需要每个处理器拥有额外的页目录，指向处理器的本地数据，并且页目录在所有处理器中保持同步。

2.2.3 IPC 机制

地址空间的引入使得各个线程运行在各自的地址空间内，从而互不干涉。但是这样一来不同地址空间内的线程也就不能协同工作。为了克服这点，内核提供了进程内通信原语，即 IPC 原语。

2.2.3.1 消息原语

对于基于消息的通信而言，至少需要两个原语，第一，发送消息至一个特定的线程，第二，从一个特定的线程接收消息[18]。

这些操作使得两个线程可以互相通信并且避免了相互间的干涉。接收者只接收自己想要接收的线程的消息。其他线程必须要等待接收者忙完手头的才能与自己通信。然而，对于提供给所有线程服务的服务线程而言，用封闭的接受原语就比较难实现，主要是以下两个原因：第一，服务线程必须要知道所有潜在的客户线程 ID；第二，由于线程在接受的时候是阻塞的，那么服务线程要么要给每一个客户线程提供一个接受线程，要么需要查询所有的客户是否有未处理的请求。为了避免这些问题，内核应该提供一个额外的原语，接收于任何线程[6]。

从多服务系统的惯常的交互模式可以发现，客户线程一般发送请求给服务线程然后等待答复。这种模式下，如果一个客户线程在发送请求之后，提出接受之

前被抢占了，服务线程就可能在客户线程提出接受之前去发送结果给客户线程。这样，服务线程就会阻塞住，不能给其他线程提供服务[24]。

为了解决这个问题，L4 提供了另外一个 IPC 原语来让客户线程原子地表达发送和接受过程。这样的一个原语就可以保证只要发送的请求消息到达服务线程那么客户线程一定是处于准备接受的状态。因此，服务器线程可以假设所有的非恶意的客户线程都是使用的原子的发送－接受原语，从而在发现一个客户线程没准备好接受时认为它是恶意的进而丢弃答复。

合并的发送－接受原语如下：

- call (发送到指定的线程并接受该线程的答复)
- reply-and-wait (发送到指定的线程并接受任何线程)
- reply-and-wait for other（发送到指定的线程并接受某一特定线程）

call 是客户线程请求服务时的典型原语，reply-and-wait 则是服务线程的典型原语。

2.2.3.2 消息类型

根据要发送的消息的大小，L4 把消息分成三种类型[24][20] [19]：

寄存器类型：短消息可以用（虚）寄存器来传递。这样的消息传递可以尽可能地避免内存访问，从而避免了 cache 和 TLB 的失配以及用户态的缺页。

字符串类型：连续的数据可以通过字符串 IPC 从发送者的地址空间拷贝到接受者的地址空间。这是消息的最常用的类型，但同时也是最慢的，因为消息至少要拷贝一次。

页面类型：如果大量的数据要传输，就可以使用虚存机制在发送者和接受者之间建立共享的内存部分。这样可以使多个地址空间之间的数据交换变得轻松，连拷贝都不需要。与另外两种消息类型不同，这种类型提供了 call-by-reference 语义，而不是 call-by-value，即，对接受者地址空间的改动在发送者的地址空间内也是可见的。如果这种情况并不是所期待的，那么这种通信就要限制接受者的访问权限。

2.2.3.3 消息时限

为了让线程能够从阻塞的通信状态恢复回来（比如与一个恶意的线程通信），每个 IPC 操作都有一个时限。第一，发送时限。发送时限所限制的是发送者发送之后等待接受者进行接受之间这部分时间。如果是 0，那么只在调用当时接受者已准备好接受的情况才会成功。否则，IPC 将立即失败，即不阻塞。第二，接受时限。接受时限限制的是一个接受线程等待发送者消息到达的时间。如果是 0，那么只有发送者准备就绪发送的情况下才会成功。第三，传输时限。通过使用传输时限，发送者和接受者都可以限制消息实际传输时的时间。传输时限是只针对字符串 IPC 的，因为所有其他类型都保证在传输过程中不会有用户态的缺页发生。由于字符串需要从发送者的地址空间拷贝到接受者的地址空间，故在这两个空间内都有可能发生缺页[3]。由于缺页是由用户态的 pager 来处理的，故不能保证完全可信，因此发送方可以指定一个传输时限来限制接受方的 pager 解决缺页时总的时间。反过来，接受方也可以指定一个时限来限制发送方的 pager 处理缺页的时间。两个时间取其小者就是最终的传输时限。

对于时限的大小，由于每个 IPC 操作都包含了多个时限（发送，接受，传输时限），时限编码必须是紧凑的。如果基于微秒的 64 位值是不能被接受的。

因此，L4 采用 16 位宽的一个数来表示时限，这个数有 10 位的数 m 表示一个因子，有 5 位的 e 表示以 2 为底数 e 为指数的值，还有一位用于区分是相对时限还是绝对时限）。这样一来，相对时限的范围就是 m*2e，即 0~1023*231 微秒（约 610 小时）[25] [26]。

对于绝对时限来说，L4 采用类似的 16 位数，4 位的指数 e，1 位的时间指示器 d，10 位的因子 m。第 16 位同样是用于区分相对时限与绝对时限。计算绝对时限还要取决于引用时钟 R，它是一个 64 位的单调递增的微秒计时器。绝对时限在以下条件成立时触发：R 的低 10+e 位等于 m << e 并且 R 的第 10+e 位等于 d。

2.3 本章小结

本章介绍了微内核的相关概念，阐述了微内核的发展历史及技术结构，简要

介绍了第一代和第二代微内核技术的发展，以及当前的代表性微内核。最后，重点介绍了 L4 微内核的设计框架，并深入阐述了它的三大组成部分，线程管理，地址空间和 IPC 机制。

第3章 用户层调度框架设计

在第一章，我们提出了微内核调度器存在的问题，L4 调度策略的实现较为简单，但如果用户想根据不同的情况，用不同的调度策略，内核是没有办法提供的。因此，用户需要在用户层实现自己的调度器。但是，这又产生另外一个问题，在内核里实现调度策略的时候，可以比较容易地获得系统资源信息。在用户层，只能获得有限的内核信息，因为在内核可以访问内核内存，但在用户层是禁止访问的，只能通过调用内核提供的接口，获得内核信息。

因此，我们设计和实现了一种调度模型，在用户层实现调度器，调度器与内核通信获得内核信息，实现自身的调度策略。

在这一章中，我们先介绍 L4 微内核现有的内核调度机制，然后再介绍我们实现的调度模型的设计和优化。

3.1 L4 微内核调度机制

3.1.1 线程优先级

内核调度器最基本的调度方式是通过线程优先级从处于准备状态的线程列表中，找到下一个运行的线程。线程最低的优先级为 0，最高优先级为 255，所有线程的优先级都处于该范围中，数字越高表示优先级越高。

线程的优先级可以在运行时刻由用户改变，但内核是不会去改变它们。所以，调度器总是选择就绪线程中优先级高的线程，优先级低的线程可能会饿死，内核不会解决该问题。

处于同一个优先级的线程，通过轮转调度方式（round-robin）来调度[27]；当时间片结束时，线程被抢占，另一个线程调度进来，被抢占的线程置于准备状态线程列表的尾部。

为了实现这些机制，L4 调度器维护着以下数据：

- 对于每一个优先级，有一个双向环形链表，包含所有就绪的具有该优先级的

线程。

- 一个包含256个指针的数组。它们分别指向每一个优先级线程队列的第一个线程（这些队列中优先级最高的队列第一个线程是正在运行的，该队列指针指向该线程）。

原则上，调度器首先检查最高优先级的那个数组项，看看是否有线程就绪了（即该数组项非空）。有的话，该线程就被选来进行分发了；没有的话，由高到低继续检查优先级队列。如果没有线程就绪的话，会分发一个特殊的空线程。

当前线程阻塞，或者耗尽了时间片的时候，同一个优先级中的下一个线程将被分发并给予其一个新的时间片。此外，指针数组中指向当前运行线程的指针更新为新选中的这个线程[27]。

时间片计算不是非常精确的，因为它是当周期性时钟中断发生时从剩余的时间片长度中减去一个时钟中断的时间长度。如果剩余的时间片变为零或者是负数，那么可称其为时间片耗尽，马上会引起调度过程。如果一个线程在两个时钟中断之间阻塞住了，下一个线程就会开始运行，这样一来，下一个线程就在下一个时钟中断之前大概只有半个中断周期的时间而不是一整个。

3.1.2 调度机制

在内核中，为了调度下一个线程，需要执行以下步骤：

- 调度器从就绪状态线程中进行选择；
- 选择优先级最高的线程列表；
- 根据轮转调度方式从该列表中选择线程。

图3.1表示调度器基于优先级调度的例子，不同优先级的线程队列处于准备状态。如果使用处理器的线程执行完毕，调度器会在优先级最高的队列选择第一个线程，作为调度的下一个线程。如果当前使用处理器的线程是时间片用完的情况，则调度器会判断该线程所在的优先级队列是否还有其他线程，如果没有该线程继续执行，如果有则把该线程置入队列尾部，并调度队列第一个线程。

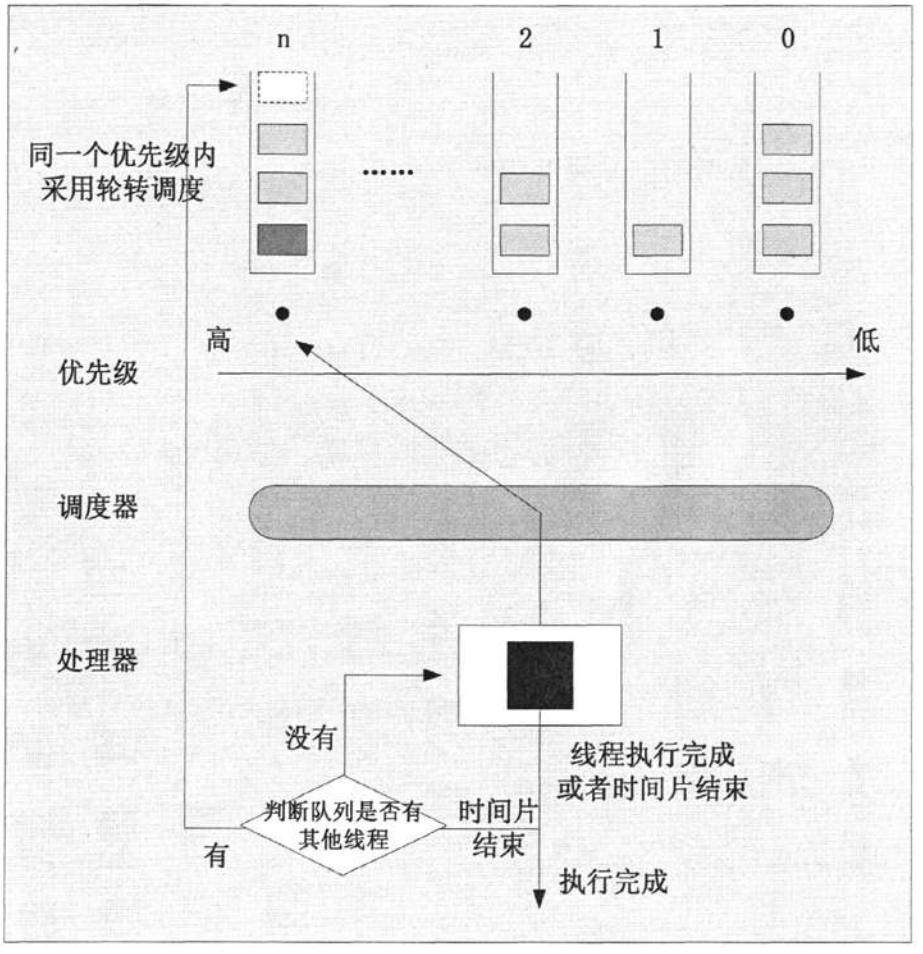

图 3.1 微内核线程调度策略

在L4中，线程的优先级通常在100左右[27]，为了快速查找到要调度的就绪线程，L4记住了就绪线程中的最高优先级，这样只要检查在这个优先级以下的那部分就可以了。当有更高的优先级的线程就绪的时候，或者在当前最高优先级无就绪线程时（该优先级的最后一个线程阻塞住了）也要更新。

L4使用了一个专门的线程来进行调度选择。这个线程在被抢占之后不会恢复，而是完全重新开始，丢弃原来所有的状态，这样同时也避免了要有一个非常大的内核栈的需要。在这个模型之下，通过调度器进行调度包含两个线程切换：第一个是从当前线程到分发器线程，第二个是从分发器线程到下一个运行的线

程。然而，由于调度器永远不会运行在用户态下，不会有一个对应的地址空间，因此切换的代价很低。由于它永远不会被用户态下的东西所引用，所以它甚至可以不需要线程id，从而可以自由地放置它的堆栈。

3.2 用户层调度模型的设计与优化

L4微内核的调度策略在上一章中已经详细介绍了，是基于优先级的时间片轮转调度，该策略实现较为简单，也基本满足用户简单的需求。但是，该策略对于一些软实时或非实时的低优先级线程，可能会出现饿死的情况。并且如果用户需要不同的调度策略，微内核是没办法提供的。

针对这种情况，在本章中，我们设计一种L4微内核上的两层调度模型，在用户层实现调度器，调度器能够获得内核系统信息，使得调度器能够按照用户的需要，实现线程调度策略或系统资源调度策略。另外，我们在用户层实现了两个简单的调度器，并对两层间的接口进行优化，提高系统性能。

3.2.1 框架设计

在用户层实现一个调度策略，调度器需要获得内核的调度数据，以及其他应用程序的调度信息。在内核里，要获得这些信息是很容易的，但在用户态，只能获得有限的内核信息，因为在内核可以访问内核内存，但在用户层是禁止访问的，调度器只能通过内核提供的有限的接口，获得内核信息。而要实现比较丰富的调度策略，调度器可能需要得到额外的调度信息，但是，现在的内核设计是不提供的。

因此，在这一节中，我们设计一个框架，使得内核和用户层调度器能够安全有效地通信，调度器能够根据策略的需要控制内核记录哪些数据，内核把这些数据记录起来，交给调度器。

3.2.1.1 总体框架

用户层调度器要实现调度策略，把处理器等系统资源分配给各个用户程序，调度器需要做到两点，第一，调度器需要收集内核资源信息，以及各个线程的调

度参数和所处状态。第二，调度器要实现自身的调度策略，进行应用线程的调度，或者系统资源的分配[28]。在这一节中，我们将设计调度器如何收集内核的资源信息，以及应用线程的调度信息。

用户层调度器要收集内核信息，可以采用两种方式，一种是通过 IPC 的方式，使得调度器与内核进行通信，这种方式的缺点是损耗了系统的性能，因为每一次系统资源改变，或者应用程序调度参数和状态改变，都需要发送 IPC 通知调度器，对性能造成影响。所以，这里采用的是另外一种机制通过内存映射的方式实现交互[29]。

我们利用 L4 微内核提供的内存映射机制，把微内核中记录内核信息的那部分内存映射给调度器，使得他们能够共享这部分内存，调度器告诉内核需要记录哪些数据，内核再把这些资源的信息和调度线程的信息，写入共享内存，从而实现了调度器与内核之间的通信，调度器获得足够的内核信息，并根据调度策略进行调度。

对于调度器收集应用线程的调度信息（包括截止时限，实时性要求等信息），我们采用 IPC 消息和地址映射相结合的方法，首先，调度器通过发 IPC 消息把控制信息发送给应用程序，通知它调度器需要哪些信息。应用程序通过两种方式把信息返还给调度器，对于一些急需的信息，用户程序通过 IPC 发送给调度器，对于其他信息，用户程序把它们写入与调度器共享的内存中，稍后再由调度器进行处理。

如图 3.2 所示，用户层调度器通过控制器来控制内核和应用程序记录哪些数据，内核记录了内核信息，并按照控制器的要求，把相关信息写入共享内存；同样，应用程序也把相关调度信息通过 IPC 或者共享内存的方式，交给调度器。调度器解释这些信息，并根据调度策略实现线程的调度。

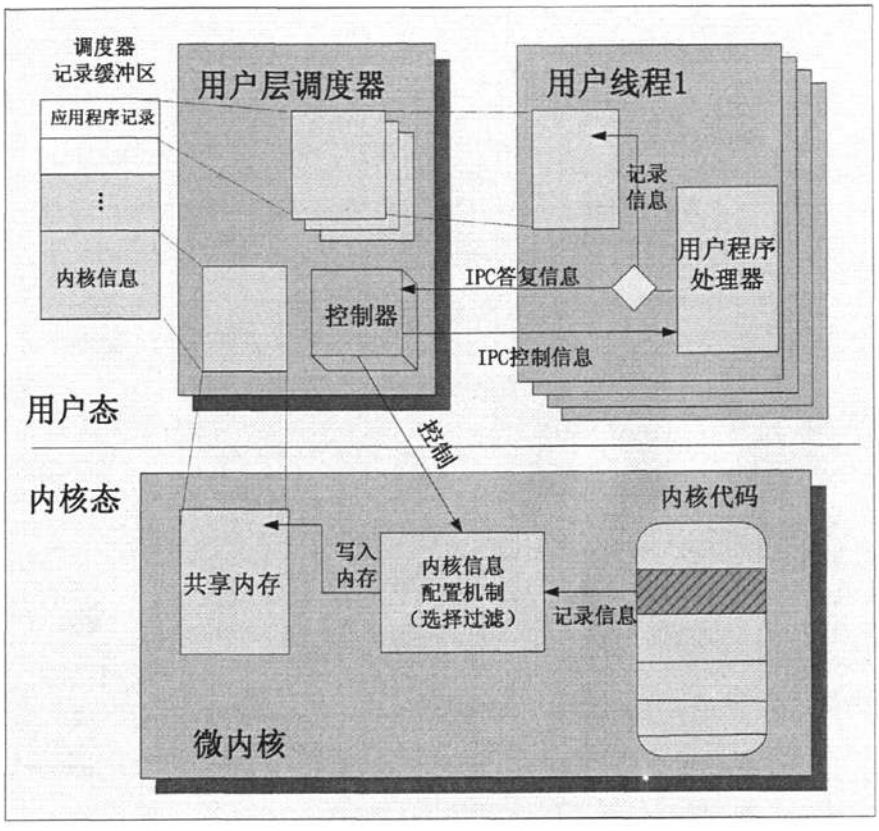

图 3.2 总体架构图

信息收集和调度器分析信息数据，实现调度策略是分开的，是两个不同的步骤，信息数据分析和调度策略实现的频率由调度器来决定的，实现频率高会提高精确度，但可能影响系统效率；实现频率低会使精确度降低。由用户调度器来决定时间和精确度的统一。

3.2.1.2 记录控制

用户层调度器通过共享内存的方式，实现了与内核的通信。在用户层，调度器可能会根据用户具体的需要，实现不同的调度策略，那么，可能每次需要记录的内核数据都会不一样。如果每次都把所有可能的数据记录下来，那么数据量可能会很大，并且很多数据用不到；或者每次换了调度策略，都要重新修改和编译内核，这明显太麻烦了，也不符合效率的要求。另外，还有安全方面的问题，如

果用户调度器能够随意获取内核信息或修改内核数据，对系统来说是一个很大的隐患，对于一个恶意的用户调度器，可能会造成系统崩毁。

因此，我们需要制定一套机制，使得用户调度器能够在运行时，动态地控制内核记录数据，获取有用信息，提高数据分析的效率。并且考虑到系统安全性的问题，还需要保证用户调度器不能在未经允许的情况下，获取内核其他信息，或者修改内核的数据。

我们在内核中设置了一些记录点，当某个事件发生时（例如新创建了一个线程或者线程就绪队列发生变化或者系统资源发生变化等等），就会跳转到记录函数，记录内核数据[30][31]。为了安全和数据分析效率的需要，内核中的事件，记录函数以及记录的数据类型都是预先定义好的，在编译的时候，触发记录动作的事件，和数据的定义就已经确定了的。

这种设计使得内核更为安全，因为记录点以及记录动作都是预先知道的，不会出现意外的情况。但是，在内核中，可能有多个记录点，用户调度器不一定每次都需要所有记录点打开来记录数据，因此，我们设计了一种动态的机制，用户调度器可以在运行时，通过系统调用，来选择内核中哪些事件的触发是有效，哪些事件无效，来达到控制记录点的目的。

另外，即使对于同一个记录点，也可能因为用户调度策略的不一样，需要记录的数据也不一样，如果每次都把所有可能涉及的调度数据记录下来，可能会有许多数据，当前调度策略是用不到的，内核记录了许多无用数据，用户调度器也需要过滤出有用的数据，这明显会降低系统性能。因此，我们设计了数据控制器，调度器通过配置数据控制器，来控制哪些调度数据要记录，达到控制内核记录有用数据的目的[30]。数据控制器存放在共享内存中，用户调度器根据需要，修改控制器结构，并把它关联到内核中某个事件，当该事件触发时，记录函数就会根据数据控制器的结构记录相关内容。

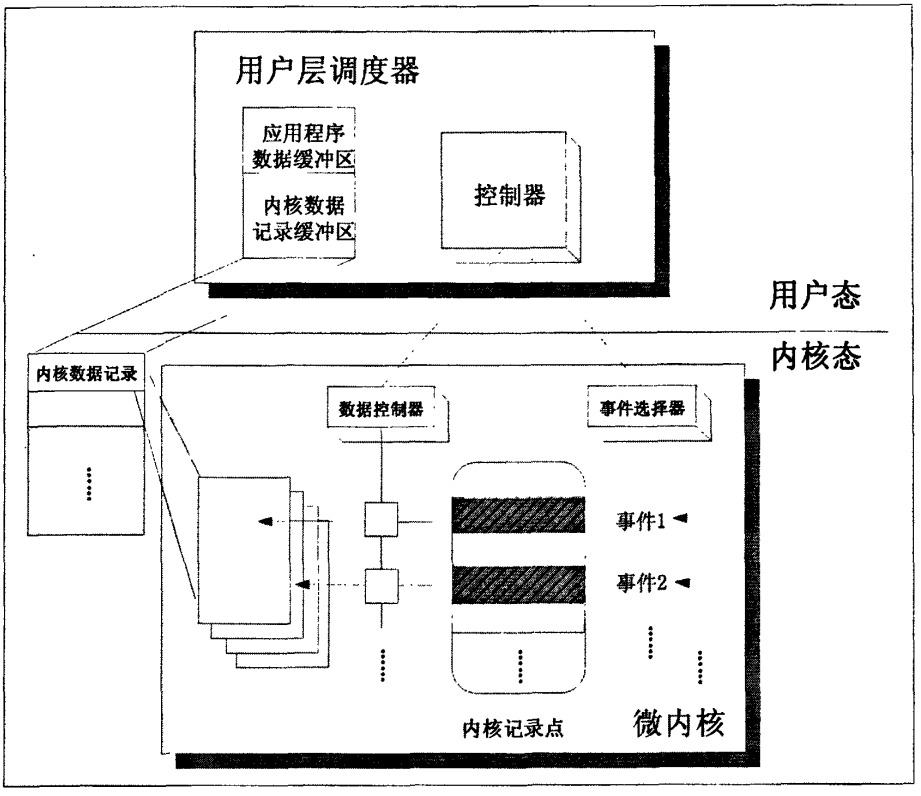

图 3.3 记录控制架构图

如图 3.3 所示，用户层调度器通过控制事件选择器，来决定内核中哪些事件打开，哪些事件关闭，控制记录点。另外，调度器也通过配置数据控制器，来决定某个事件触发时，内核记录哪些数据。

对于用户线程，由于与调度器同在用户态，所以，可以直接利用 IPC 通信方式，来实现调度器对用户线程记录数据的控制。

3.2.1.3 读写冲突处理

用户层调度器和内核共享内存，用户层调度器读数据，内核写入数据，它们之间可能会发生读写冲突，如果调度器正在读取数据，而内核正在写入信息，或者内核正在进行信息写入动作但还没有完成，而调度器开始了读取数据，这两种情况可能会造成数据的不一致[32]。

为了解决这种情况，我们对于一个记录信息的内存页用两个有效位，三个状

态来处理这种读写冲突。当内核正在写入数据时，我们通过该位禁止调度器读取信息，直到内核写入完成，而当调度器正在读取数据时，我们允许内核写入数据，调度器读取完数据后，发现它在读数据时，内核开始了新一轮的写入信息，因此，调度器等待内核的信息写入完成，然后重新开始读取数据，再进行数据分析。所以，内核可以在任何时候写入数据，不需要看该位的值，而调度器需要根据该位的值来决定什么时候读数据。

3.2.2 用户层调度器

3.2.2.1 完全调度器

在 L4 微内核架构上，根据上一节的框架，我们设计和实现了一个用户层调度器，完全取代内核调度器。该调度器主要用到 EDF 算法，该算法是一个动态的调度算法，当需要进行调度的时候，调度器会寻找下一个截止时间最短的就绪线程，进行运行。如果有一列就绪线程，这些线程都有以下调度参数，最坏执行时间（用 C 表示），截止时间（用 T 表示），则 C/T 表示一个线程占用处理器的宽带，那么，这些线程必须满足以下公式：

$$\sum_{i=1}^{n}\frac{C_i}{T_i}\leq 1,$$

才能使得这些线程能够在规定时间内完成，因此，在 EDF 调度器上层，我们设计了一个准入控制器，根据上面的公式对就绪线程进行准入控制，截止时间短（T 较小）的线程依次通过控制器，直到下一个线程如果通过，$\sum_{i=1}^{n}\frac{C_i}{T_i}>1$，那么，不让该线程通过，这时处理器宽带已满[34] [35]。

该算法比较简单，并且能够较好实现实时性和公平性，低优先级不会发生饿死的情况，但高优先级线程与低优先级线程相比，没有优势。因此，我们在设计的时候稍作优化，在准入控制器上，增加一个参数预处理器。该预处理器能根据优先级高低对截止时间作相应调整，高优先级的线程，通过预处理器时，它的截止时间会变得较小，体现它对低优先级线程的优势。

L4 微内核一共有 256 个优先级，一般线程的优先级在 100 附近，在我们设计

的预处理器中，简单把线程优先级分成高中低三种，这几种线程截止时间的处理如下所示：

优先级大于 110 的线程，$T_i = T_i / 1.2$；

优先级大于等于 90，小于等于 110，$T_i = T_i / 1.1$；

优先级小于 90 的线程，$T_i = T_i$。

综上所示，该调度器包括三个部分，参数控制器，准入控制器，以及 EDF 调度器，如图 3.4 所示。

调度器从内核中获取就绪线程列表，并且从各个用户线程获取需要的调度参数（包括最坏执行时间，到达时间，截止时间，线程优先级等）。就绪线程通过参数控制器，进行参数预处理，通过准入控制器，进行准入控制，然后根据 EDF 算法进行调度，找到下一个要运行线程。

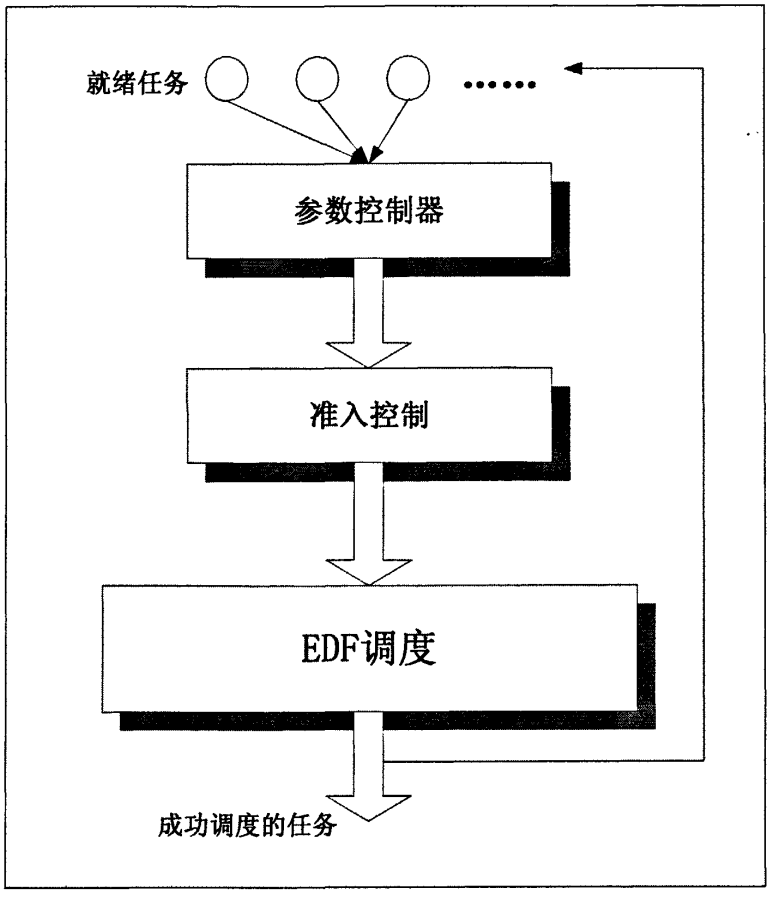

图 3.4 完全调度器流程图

该算法较为简单，并且通过参数控制器进行参数设置，能够实现线程实时性和公平性的统一。

3.2.2.2 辅助调度器

如上一章节所说的，L4 微内核实现了基于优先级的轮转调度算法，该算法实时性强，实现较为简单，但是该算法会导致饿死现象，如果一直有高优先级的线程处于就绪状态，那么，低优先级线程永远不会执行。因此，在用户态可以实现一个辅助调度器，来优化微内核调度器的调度算法，主调度算法还是微内核的基于优先级的轮转调度算法，具体的调度也还是在内核里实现。

该辅助调度器的主要作用是防止低优先级线程发生饿死的现象，具体的算法

是，每隔一段较长时间，如果处于就绪状态的线程还没有被调度的，那么，该线程优先级加 1，直到该线程被调度为止。

如图 3.5 所示，在微内核内，内核调度器每次选择了一个就绪线程在处理器运行，就会在共享内存里记录选中线程的相关信息（包括线程 ID，时间片长度等），用户层辅助调度器通过一个定时器，每隔一段较长时间就会读取这些信息，因此，调度器知道在这段时间里，哪些就绪线程运行了，哪些就绪线程没有运行，调度器根据自身策略，通过调用 schedule 系统调用，对就绪线程的相关调度参数进行修改。

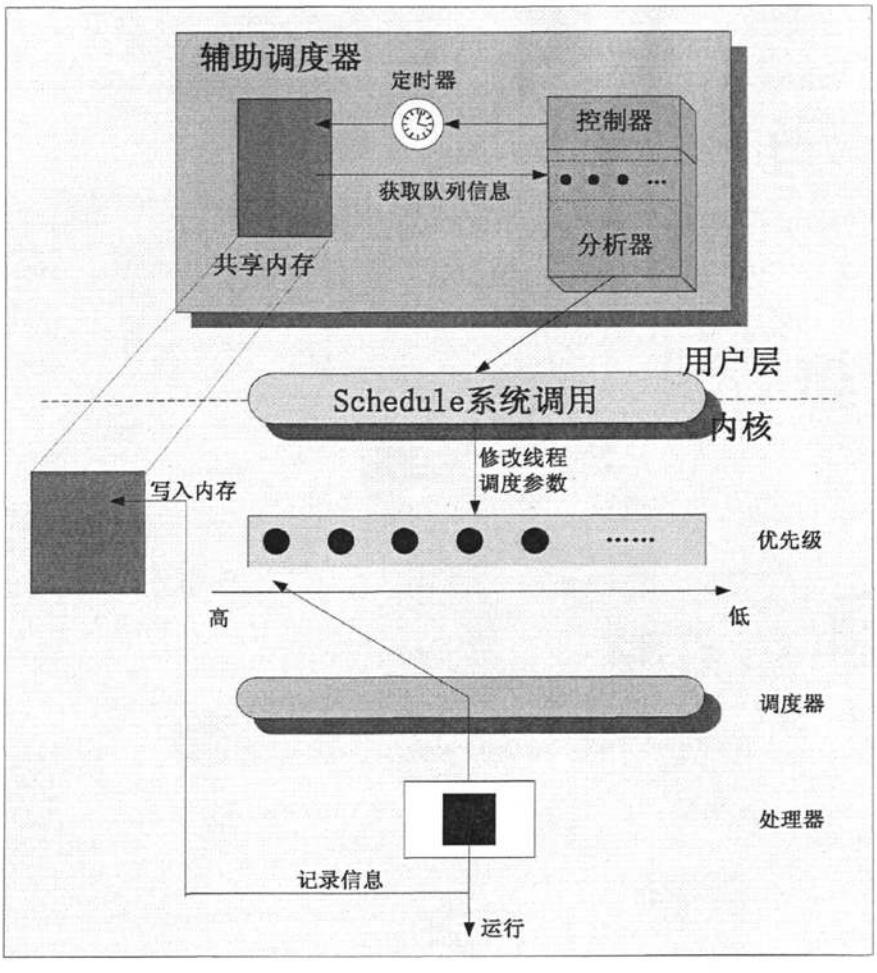

图 3.5 辅助调度器流程图

3.2.3 接口改进

在3.2.2.2节，我们在用户层实现了辅助调度器，调度器通过调用schedule系统调用，修改线程的优先级，时间片长度等参数，来优化微内核中的调度策略，达到防止线程饿死的目的。L4微内核提供的schedule系统调用，每次只能修改一个线程的调度参数，而辅助调度器经过调度策略的计算，可能在同一时间需要批量修改多个线程的调度信息，需要多次调用schedule系统调用，这会导致频繁进出内核，系统性能会大大降低。

为了解决这个问题，使得用户层调度器能够通过一次进入内核，批量修改线程的调度参数，我们在兼容原有接口的前提下，重新设计了schedule系统调用接口，并修改了其在内核中的实现方式。

下面我们首先介绍L4中schedule系统调用的接口实现，及各个参数表示的意思，然后介绍经过我们改进的接口。

3.2.3.1 系统调用接口

L4微内核为用户提供了schedule系统调用，该系统调用可以被调度者用来定义优先级，时间片长度，和其他线程的调度参数，它甚至可以确定线程的状态。当且仅当调用该系统调用的线程被定义为目标线程的调度者，这个系统调用才能起作用。下面介绍原schedule系统调用及输入参数，输出结果，该系统调用的原型为[20][36]：

word Schedule（ThreadId dest, word time control, word processor control,
word prio, word preemption control）

输入参数[20]：

- dest：目标线程ID，目标线程必须存在（但可以是不活动的），并且当前线程必须定义为目标线程的调度者。否则，目标线程没有用。
- time control：目标线程新的时间片长度，不可以为0，但可以是无穷大，在这种情况下，线程不会因为时间片耗尽被抢占。时间片长度最小单位为1微秒（1 μs），所以，该参数的值（可能不为整微秒）总是接近于可能的时间片长

度。

- prio：目标线程新的优先级，必须小于等于当前线程优先级。
- preemption control：该参数包括 8 位的 sensitive prio 域和 16 位的 maximum delay 域。
  - sensitive prio：如果抢占线程的优先级小于该参数，（a）如果 delay-preemption 参数被置位了，则会延迟直到线程调用 thread switch 系统调用。（b）如果 signal-preemption 被置位了，则产生一个抢占异常给异常处理句柄。如果抢占线程的优先级大于该参数，则不会产生抢占延迟或者发送抢占异常给异常处理句柄（不用考虑 delay-preemption 和 signal-preemption 参数）。
  - maximum delay：延迟抢占发生的最长时间
- processor control：该参数后 16 位有用，说明线程在哪个处理器执行，这个参数必须是有效的，必须小于处理器总个数，否则，该参数被忽略。第一个处理器为 0 号处理器，其他类推。

该系统调用输出结果如下[20]：

- tstate（Thread state）

  0 error，操作失败，ErrorCode 中说明失败原因。

  1 dead，线程不能执行或者不存在。

  2 inactive，线程不活动的或者停止了。

  3 running，线程已经准备好在用户层执行。

  4 pending send：用户调用的 IPC 操作正在等待接收者准备好接收信息。

  5 sending，用户调用的 IPC 操作正在执行发送信息 IPC 操作。

  6 waiting to receive：用户调用的 IPC 操作正在等待接收信息。

  7 receiving，用户调用的 IPC 操作正在执行接收信息操作。

- ErrorCode: 如果参数运行结果低 8 位为 0，则定义，否则不定义

  =1 没有权限，当前线程不是目标线程的调度者

=2 dest 参数表示的是一个不合法的线程 ID

=5 调用过程中有不合法参数（时间片长度，优先级，处理器号等）

- time control：该结构 32 位，前 16 位为 rem ts，表示当前时间片剩下的时间，后 16 位为 rem total，表示线程剩余的总运行时间。

3.2.3.2 接口改进

在这一节中，将会阐述设计方案，修改 L4 微内核的 ABI/API 接口，为用户提供一个新的 schedule 系统调用。该系统调用接收批量线程 ID，一次性进入内核，对线程的调度参数进行修改，并选择线程进行调度。并且我们需要尽量兼容原有的代码及接口[37]。

（1）系统调用接口

原系统调用每次处理一个线程，所以只需要一个线程 ID 作为参数。我们修改的系统调用一次需要处理多个线程，需要把这些线程 ID 告诉内核，因此，我们需要增加两个参数，一个是处理线程 ID 的列表，一个是处理线程的个数。定义的格式如下所示：

```
Word Schedule ( ThreadId* dest, Word* n, Word TimeControl, Word ProcessorControl,
                Word Prio, Word PreemptionControl, Word* result, Word* old_TimeControl )
```

在原来系统调用中，参数都放入寄存器传递给内核，新系统调用需要把多个线程 ID 传递到内核，按照这个办法明显是行不通的。因此，我们利用线程 UTCB 中的消息寄存器，把线程 ID 放入消息寄存器中，传递给内核，内核处理的时候从消息寄存器中读出线程 ID，如果每个线程的调度参数不一样，也按顺序放入消息寄存器中。返回结果也会遇到同样的问题，所以，当完成后，我们把返回结果也放到消息寄存器中[38]。线程的消息寄存器只有 64 个，因此，一次系统调用不能超过 64 个线程，如果需要传递其它调度参数的信息，每次该系统调用可以处理线程的个数也会相应减少。

如图 3.6 所示，表示原 schedule 调用时，各个参数放入相对应的寄存器内，传递到内核。

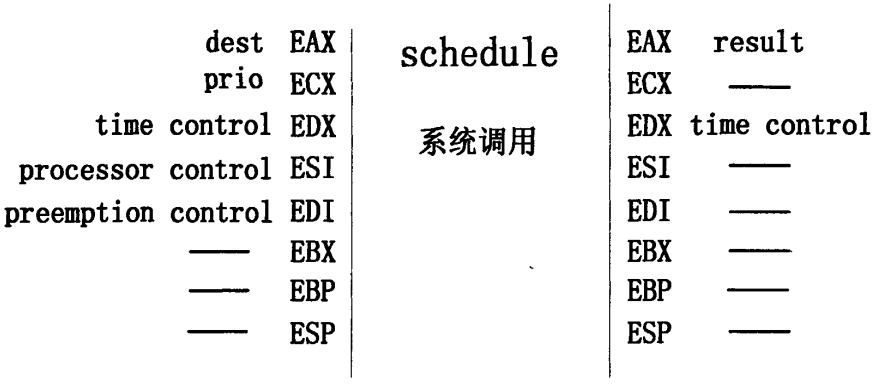

图 3.6 schedule 系统调用二进制接口

调用新 schedule 系统调用时，我们需要把 UTCB 的地址告诉内核（内核也可以通过线程 ID 找到该 UTCB），省却了查找的时间，所以，在定义二进制接口时，我们需要把处理线程的个数 n 放入寄存器，把 UTCB 的地址也放入寄存器，另外，如果修改的线程优先级（即 prio 参数）都一样，也放入寄存器，否则存入 null，各线程优先级从消息寄存器里读取。

（2） 通用编程接口

在 L4 中，从 schedule 系统调用衍生出四个通用编程函数，提供给用户编程编程使用，它们分别是 SetPriority（）函数，调用该函数改变线程优先级；SetProcessorNo（）函数，调用该函数改变线程运行的处理器；SetTimeslice（）函数，调用该函数改变线程的时间片长度；SetPreemptionDelay（）函数，调用该函数改变线程的延迟抢占发生的最长时间。

对于这些通用编程函数，我们通过 C++重载的方式，增加这些编程接口重载版本，使得它们可以一次处理多个线程，并且兼容原版本的编程接口，如图 3.7 所示，新的 scheduleNThread 接口，除了支持重载的编程接口，还支持原来的接口，通过在原来的接口的代码中，调用新的接口来完成实现。

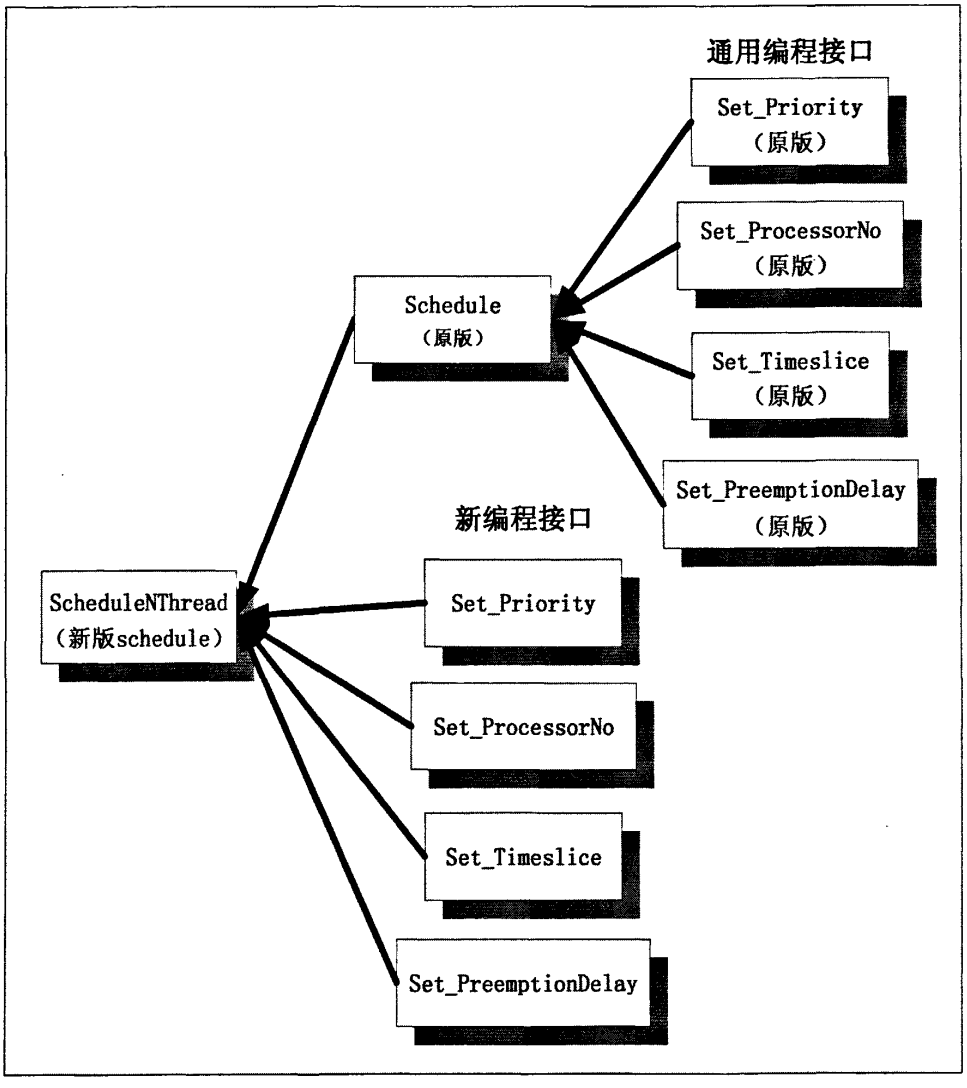

图 3.7 新通用编程接口

3.3 本章小结

本章设计和实现了一种基于 L4 微内核的两层调度模型，在用户态实现调度器，调度器获得内核系统信息，和应用线程调度信息，实现自身的线程调度策略或系统资源调度策略，并且调度器通过记录控制器控制获取哪些数据。另外，本章根据设计的框架，在用户层设计了两个简单的调度器，一个是以 EDF 算法为基础的调度器，完全取代内核调度器，在用户层实现调度功能。另一个是实现了辅

助调度器，对内核的调度算法进行改进利用，主要调度功能的实现还是在内核中完成，证明了能根据设计的框架，在用户层实现调度器。再者，本文改进了一个系统调用接口，提高了用户层调度器通过这个接口进入内核的性能。

第4章 用户层调度框架实现

在上一章，我们介绍了两层框架模型的设计，以及接口的优化处理，在这一章，我们将具体阐述如何实现。

4.1 框架实现

用户层调度器要顺利执行调度策略，需要获得内核的系统信息以及其他应用模块的调度信息，因此，需要设计一个架构使得调度器与内核，调度器与应用程序实现通信，调度器能够控制记录事件以及记录的数据。

4.1.1 通信机制实现

4.1.1.1 微内核与调度器通信

用户层调度器需要与内核交互消息，获得内核调度信息和资源信息，可以采用把内核信息记录在内核内存里，再把这部分内存拷贝到调度器内存，这需要进行内存拷贝，对性能影响很大，特别是操作比较频繁的时候。在我们设计的系统里，采用了另外一种方法，利用L4提供的内存映射机制来使内核与调度器通信，我们分配一片固定的内存给内核记录调度信息和资源信息，再利用映射机制，把这部分内存直接映射到调度器的地址空间里，使得微内核和调度器能够共享这部分内存[39][40]。由于调度信息比较小，所以共享内存是比较有效和节省空间的方法。

以下我们先介绍 L4 的映射机制，然后说明如何利用该机制实现内核与调度器共享内存。

（1）L4 映射机制

原则上，只要另一进程允许，一个进程就可以利用授权和映射调用，把另一进程地址空间中的一个页面映射到自己的一个页面上，从而方便这两个进程可以同时访问一个页面，即实现数据的共享与交互。

地址空间管理原语负责内存地址空间的映射。初始化时，创建一个特权线程sigma0，它是一个运行于用户层的服务线程。物理地址全部赋予该线程，以后对

其他线程内存的分配管理由 sigma0 线程负责。

对地址空间的操作方面需要支持三个原语：Grant、Map 和 Unmap，即授权、映射、解除映射。基于这些操作，微内核可以支持由不同的用户进程以不同的策略来映射页面。

- Map（映射）：将本地址空间的指定部分映射给另一个线程，从而让另一个线程能够访问到本线程的地址空间。映射之后，本地址空间仍然能够访问这部分空间，即，此刻双方都能访问对应空间。实现的是一种共享的模式。此外，必须支持两种映射模式：
  - ✧ 给予读权限与写权限的映射；
  - ✧ 只给予读权限而没有写权限的映射。
- Grant（授权）：其他方面同 Map 一样，不同的是，当前地址空间中的相应的空间将被除去，即进行授权之后，本地址空间不能再访问这些已被授权出去的部分。对该部分地址空间的管理权限完全地授予另一个线程。
- Unmap（取消映射）：执行过 Map 操作的线程可以通过 Unmap 操作来取消映射出去的地址空间；或者在之前将读与写权限都映射出去的情况下通过此操作（结合一定的参数）来取回写的权限，而保留被映射至的线程的读的权限。

在这三个原语之上，还要遵循一个原则，即地址空间必须支持递归地映射或授权机制。即 A 线程将某部分地址空间映射（或授权）给 B 后，B 可以将这部分中的一部分或全部再映射（或授权）给 C 线程，以此类推。同样，也需要支持逆向地递归取消映射。

地址空间管理的示意图如图 4.1 所示：

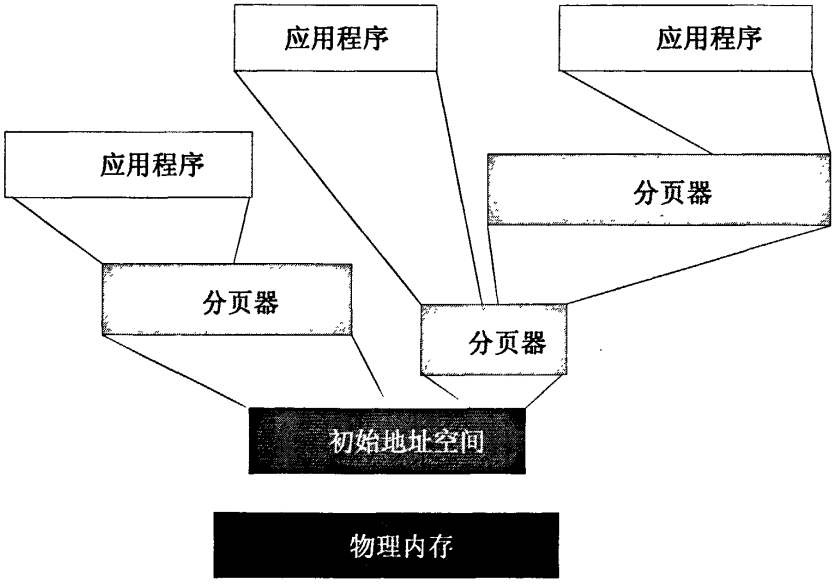

图 4.1 地址空间映射示意图

（2）内核与调度器共享内存

在内核中，我们静态分配一片固定的内存，当内核需要记录数据时，就记录在该片内存里。

内核同时把这部分内存区域交给用户空间的特权线程sigma0，sigma0再用L4的内存管理原语，把这部分内存映射给调度器线程，调度器就能读取记录数据，并进行分析。

由于调度器需要对内核记录哪些信息进行控制，所以，我们在内存中添加了控制记录信息的数据结构，调度器通过改变这些结构的具体数据，来达到控制内核记录的目的。因此，调度器也需要对这部分内存有写权利，进行内存映射时，必须把这部分内存映射为可写的[41]。如图4.2所示，内核申请了一小片固定内存，与用户层调度器共享调度信息，并把这部分内存交给sigma0，sigma再把内存交给调度器。

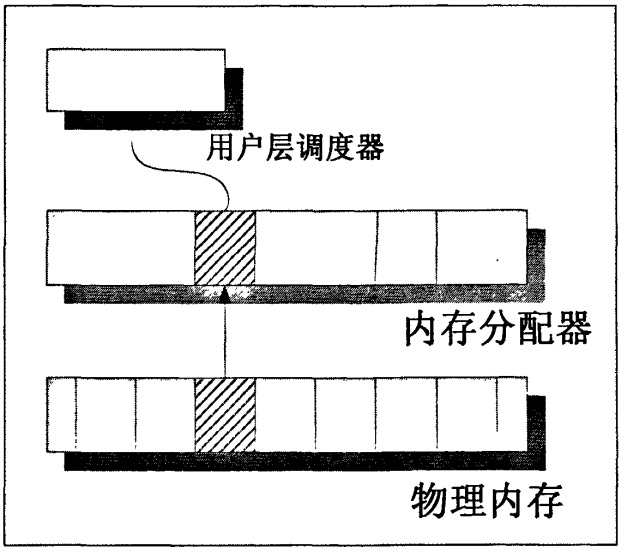

图 4.2 内存分配示意图

但是，这种设计可能意味着一个恶意的用户调度器可能会通过修改控制数据，来做一些非法的事情，可能会影响整个系统的安全。为了解决这个问题，我们必须保证一个被恶意修改的数据控制器不会影响到系统其他部分以及整个系统的安全。这样，我们在实现的时候必须做到两点，第一，记录控制结构不会在内核代码其他地方用到，这个很容易实现，控制的结构只是用来记录数据的，不会修改其他内核代码。第二，内核记录信息的时候不能越界。由于内核记录数据的位置是根据数据控制器来指定的，一个被恶意修改的控制器，可能会使记录位置越界，产生安全隐患。我们在实现的时候保证记录的位置在共享内存那一范围内，因此，在记录时，我们让内核先检查一下记录位置有没有越界，然后才进行写入内存。

4.1.1.2 调度器与应用程序通信

由于调度器与应用程序都在用户层，因此，我们并不需要像控制内核那样自定义一套机制来实现与应用程序的通信和控制，我们利用 L4 提供的 IPC 原语，来使得调度器与应用程序实现通信。调度器通过发 IPC 消息把控制信息发给应用程序，通知它调度器需要哪些信息。程序可以通过两种方式把信息发送给调度器，

对于一些急需的信息，程序通过IPC形式发送给调度器，在调度器那里进行解释，调度器可以立刻知道程序的信息发生了改变，执行相应的策略，这种方法比较实时，但是大量的IPC操作有可能损耗系统性能。另外，我们同样可以利用地址映射机制，把调度器作为应用程序的分页器，当应用程序需要写入调度信息而发生缺页时，调度器把记录程序信息的内存页映射过来，并且告诉程序写入哪个位置，然后程序更新信息。对于大量的信息，或者不是急需的信息，采用这种方法较为合适。但这种方法对信息的修改可能不能立刻让调度器获知，因为调度器不一定实行轮询的机制，可能采取过一段时间检测一次的机制，所以可以采用两种方法结合的办法，对于一些急需的信息，用户程序通过IPC发送给调度器，其他信息程序把它写入内存中。

4.1.2 事件机制实现

上一节介绍了如何使得内核与用户层调度器通信，内核记录的调度信息能交给调度器。我们通过在内核中设置事件，当某个事件发生时，内核中对应的记录点就会调用记录函数，记录内核信息。我们设置的记录点以及定义的记录函数都在编译时就确定的。对于不同的调度策略，可能需要不同的事件，发生记录的位置也不一样。因此，在内核中，可能有多个记录点。如果这些记录点都打开，那很明显，记录的无用数据会有很多，并且也影响了性能。所以，我们设计了一种动态的机制使得用户层调度器能够在运行时选择关闭或打开记录点。

下面将会介绍实现这种动态策略的方法。

第一种方法，用条件判断的方法，每次运行到记录点位置，判断该记录点是否打开，记录点打开就跳到具体记录函数。我们在内核中定义一个事件数组evt[]，evt[0]表示记录点0的开关，如此类推，该值为0表示关闭记录点，1为打开。初始化的时候该数组都置为0，用户层调度器通过系统调用来改变这些记录点开关的值，从而实现打开或者关闭记录点。

上面那种方法比较简单直观，但是，每次运行到记录点的时候都要对该记录点的参数进行判断，会影响系统的性能。因此，我们采用另外一种动态指令的方

法。

在内核代码中，每一个记录点，我们插入5个字节的nop指令，来为jmp指令预留位置，当某个记录点打开了，那么，该地方就会插入一个jmp指令，取代原来的nop指令，目标位置是该点记录函数的位置，当内核指令运行到这个地方，就会跳转到记录函数进行记录[42]。

在内核中，除了需要定义好目标指令以外，还需要保存插入的内存位置，即需要记录下这些nop指令开始地址，以及目标函数的地址。当记录点打开时，在nop指令的地方替换为jmp指令，跳转到记录函数。指令替换的位置在编译时已经知道的，在运行时就能动态地进行替换。

内核增加一个系统调用接口，系统调用包括一个记录点ID参数，用户调度器能够在运行时通过该接口，传送一个记录点ID，来达到动态启动和关闭该记录点的目的，一旦调用该系统调用，内核会按照上面的原则动态地改变目标地址的指令。

初始化时，所有的记录点都是关闭的，在每一个记录点都只是多执行了一些nop指令，对系统性能的影响是很小的，相比于在方法一中，每次运行到记录点都需要判断参数，这种方法更有效率。

4.1.3 控制及记录数据

在上一节中，我们实现了事件机制，当某个事件触发时，就会在该记录点跳转到记录函数，进行内核信息的记录。用户调度器可以在运行时刻，启动或者关闭记录点，来控制在内核哪个位置记录信息。但是，即使在同一个记录点，调度器也可能因为实行的调度机制不一样，所需要的内核信息可能也不一样[43]，怎么控制内核记录哪些数据呢？为了达到控制记录数据的目的，我们增加一个数据控制器，用户通过配置这个控制器，来决定当某个事件发生时，记录点位置记录哪些内核数据。

4.1.3.1 定位数据控制器

即使对于同一个记录点，用户可能由于实行的调度策略不一样，需要的信息

也不一样，所以，用户应该能够在运行时，选择在哪个记录点用哪个数据控制器，来记录内核数据。为了达到这个目的，在共享内存中，我们通过维持一个对应关系表格，来实现这种对应关系，在每一个内核记录点，都知道该点的ID，记录函数通过查找共享内存中记录点ID和控制器ID对应表格，找到该记录点的数据结构，然后按照该数据结构的格式，来记录内核信息。用户层调度器通过配置或者改变该记录点对应的控制器，来达到控制该位置记录哪些内核数据的目的。具体实现过程如图4.3所示，记录函数通过该点的ID找到对应的控制器ID，再通过基地址找找到具体的控制器结构。

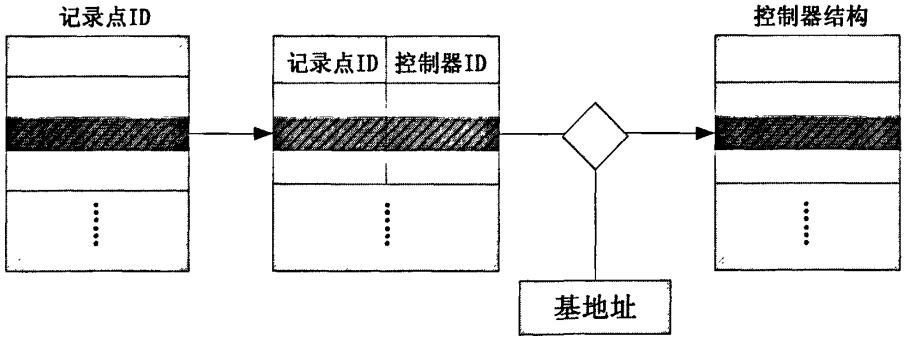

图 4.3 查找数据控制器流程图

4.1.3.2 数据控制器结构

在上一节我们介绍了如何关联记录点和数据控制器，在这一节我们将介绍控制器的具体结构。

在我们设计的模型里，一个记录控制器由32位组成，大致可以分为三个部分，第一部分是当前记录位置的偏移，该部分的作用是指明数据应该记录在共享内存中哪个位置。第二部分表示各个参数的设置（如果某个参数置为1，表示要记录该数据，否则不需要记录）。因为对调度器来说，对于某个调度策略，可能只是需要记录其中某些参数，所以只要把控制器中这些参数置为1，其他置为0。第三部分是上次记录信息的大小（通过当前记录位置及上次记录信息大小，用户层调度器可以读出上次记录的信息，然后进行分析处理）。

以下以我们设计的控制器结构作为例子，说明该控制器是如何控制内核记录

信息的。

在3.1节设计的第一个算法里，需要记录内核当前就绪线程队列的长度，各个就绪线程的ID，如果线程就绪队列发生改变，需要通知用户层调度器，因此，我们在该点设置事件，当线程就绪队列发生变化时，就会进入该记录点，调用记录函数。我们根据这个记录点的ID找到它对应的记录控制器，如图4.4所示，控制器一共有32位，前16位表示偏移，中间7位表示各个参数，最后9位表示记录的大小。

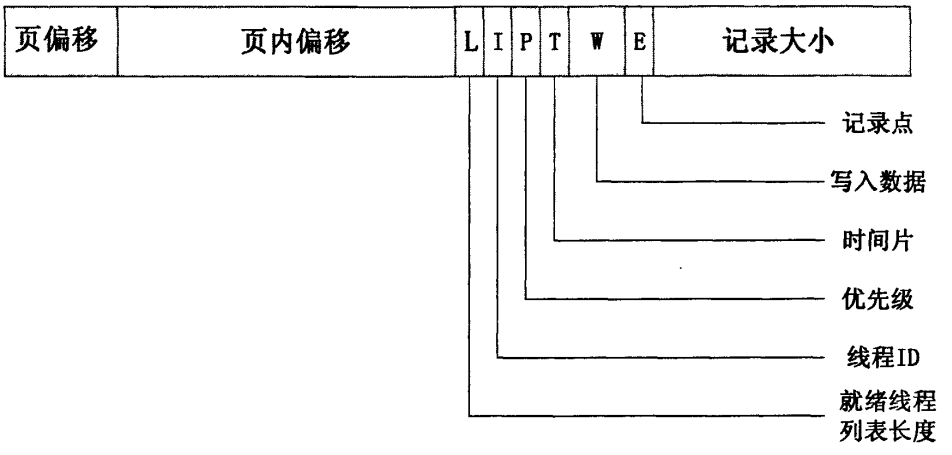

图 4.4 数据控制器结构图

以下分别介绍各部分的作用。

控制器中偏移有16位，所以，共享内存的大小为64KB（这个不是必须的，可以根据自己的需要调整控制器的大小，调整偏移的位数，从而调整共享内存的大小）。在L4内核中，一个页的大小为4KB，所以，记录区域共包含16页，控制器偏移的前面4位代表这16页中哪一页，一个记录控制器对应一个内存页，根据这个控制器记录的内存数据都放在该页中，因此，同一个控制器前四位总是一样的，偏移的后12位表示在这个页中当前的记录位置，如果要记录信息就在这个位置进行记录。当该页记录满的时候从头开始记录，覆盖掉原记录，因此，如果用户层调度器需要保留以前数据，需要在自己的地址空间另外保留一份。数据控制器中偏移的结构如图4.5所示。

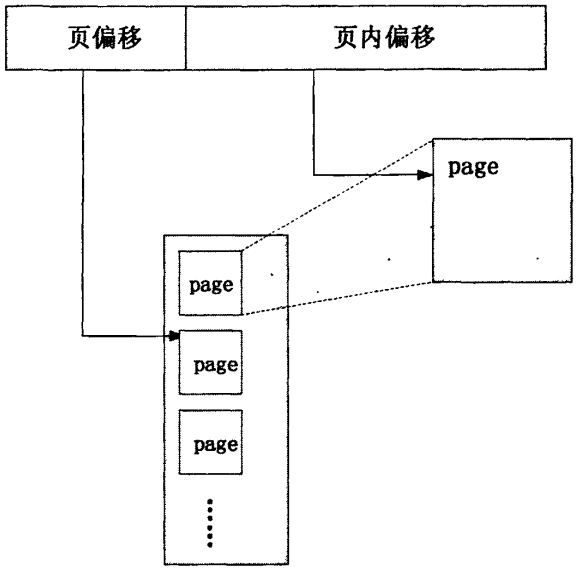

图 4.5 记录内存定位示意图

记录控制器最后9位表示最新一次记录的大小，用户可以根据当前记录位置的偏移以及最新一次记录的大小，找到最新一次记录的开始位置，从而读出记录信息。

记录控制器中间7位表示各个参数，具体说明如下：

L位，表示是否要记录列表长度；

I位，表示是否要记录各个就绪线程ID；

P位，表示是否要记录各个就绪线程优先级；

T位，表示是否要记录各个就绪线程时间片长度。

W位，由两位组成，表示内核是否写入了数据，该位也用来处理同一个内存页中，用户层调度器和内核间的读写冲突。

E位，表示进行记录的记录点ID。

对于前面四位，如果置为1表示要记录，如果为0表示不需要。用户层调度器通过这个控制结构，记录它所需要的信息。对于W位，将在下面详细介绍该位如何处理读写冲突。

4.1.3.3 记录和读取数据

在内核中，记录内核信息的流程如图4.6所示，当内核代码运行到记录点的时候，跳到记录函数，记录函数根据记录点ID找到相对应的记录控制结构，根据控制结构来记录内核信息。首先，根据结构中的内存页偏移，找到内存页，然后根据当前记录位置，找到内存页中的记录位置，记录新的内核信息就从该位置开始。然后按照控制结构中各个参数进行相关记录，把信息写入内存中，完成写入后，更新控制器，主要更新当前内存页记录位置，返回信息，和记录大小，最后，退出记录函数，完成记录动作。

以上一节中设计的控制器为例子，我们需要记录就绪线程长度，线程列表，线程优先级列表，时间片长度列表。找到记录位置后，先写入发生写入动作的记录点ID，然后是就绪线程队列长度，判断控制结构中I位是否为true，如果是则记录各线程ID，在I位为true的前提下，判断P位和T位，如果P位为true，记录各线程优先级，如果T位为true，记录各线程时间片长度，然后填充位数，使得记录最好是4字节的倍数，信息记录完成后，再更新控制器中当前记录指针的位置，并且根据当前记录指针和开始时记录的位置，计算出这次记录数据的大小，然后把该记录大小更新到控制器的size域。

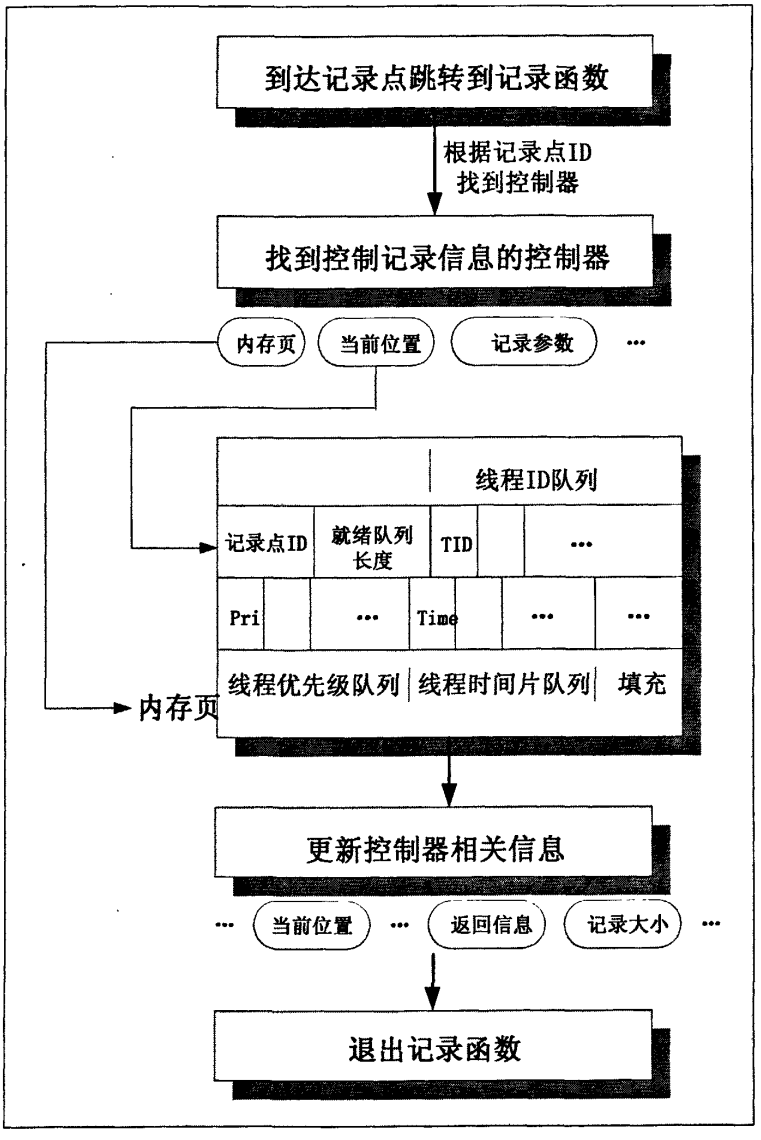

图 4.6 记录数据流程图

在用户层，调度器读取内存中记录的相关数据内容，根据数据控制器，算出最新一次记录的开始位置，然后先读取产生这次记录的记录点ID，然后根据控制器各位参数，读出具体内核信息。

在用户层调度器，储存线程调度信息的结构为：

```
struct thread_schedule{
    threadID;   //线程ID
    priority;   //线程优先级
    timeslice;  //线程时间片长度
……
};
```

从内存中读出数据的相关代码为：

```
int read_record()
{
    ……
    pos = controller.position - controller.size;   //得到记录位置
    recordID = record_area[pos++];    //得到记录点ID
    if (controller.length == TRUE)
        length = record_area[pos++];   //得到就绪线程长度
    if (controller.threadID == TRUE){
        for(int i=0; i<L4_length; i++){
            thread_user [i].threadID = record_area [pos++];
            //得到各就绪线程ID
        }
    }
    if (controller.piority == TRUE){
        for(int i=0; i<L4_length; i++){
            thread_user [i].piority = record_area [pos++];
            //得到各就绪线程优先级
        }
    }
    if (controller.threadID == TRUE){
        for(int i=0; i<L4_length; i++){
            thread_user [i].timeslice = record_area [pos++];
            //得到各就绪线程时间片长度
        }
    }
    ……
}
```

4.1.4 处理读写冲突

设计记录结构时，我们规定一次记录的数据不超过4KB大小，所以，一次记录不会覆盖自身的数据，但有可能覆盖前一次记录的数据（如果用户希望保持以前的信息，需要另外保存在自己的地址空间）。这时候，可能会产生一个问题，当用户层调度器读取内存数据时，内核又写入新数据，并且新数据比较大，把原数据覆盖了，而调度器还没读完数据，它读到的一部分是旧数据，一部分新数据，造成前后数据的不一致。为了解决这种情况，在数据控制器中，有两位（图4.4中的W位）用来处理读写冲突。初始化时，该位为00，当内核开始写入数据时，把该位写为01，当内核完成写入数据时，该位写为11。在用户层调度器，如果发现该位为00或01，都不会读数据，如果为11，表示已经记录了新的内核数据，因此，调度器把该位写为00，并开始读取数据。如果发生上面所说的冲突，即调度器读取内存数据时，内核又写入新数据，那么当调度器读完数据时，会发现该位已经发生变化，说明数据发生了变化，调度器重新读取数据。另外，调度器也可以根据读数据前和读完数据后，控制器前后两次的页内偏移是否发生了变化，来判断这段时间内，内核是否写入数据。

4.2 用户层调度器

按照上两节的方法，我们从内核和应用程序中获取了调度信息，然后在用户层调度器实现具体算法选出下一个运行线程。

表4.1是用户层调度器用到的部分L4微内核提供的接口函数[33][20]。

表 4.1 微内核接口函数

L4 接口函数	作用
L4_GlobalId	获取一个线程的全局 ID。
L4_Wait	等待任意一个线程发送给它 IPC 消息。
L4_Send	给指定的线程发送一个 IPC 消息。
L4_Receive	接收到一个信息。
L4_ReplyWait	等待接受来自任意一个线程的 IPC 消息并给指定线程发送一个回复消息。
L4_Load	把一个消息对象的值加载到消息寄存器。
L4_Set_Label	为一个 IPC 消息设置标签。
L4_Clear	清空一个消息对象 L4_Msg_t
L4_ThreadSwitch	线程切换，把 cpu 交给指定线程。
L4_ThreadControl	创建或删除一个线程，或者修改一个已存在线程的全局 ID。
L4_Schedule	这个系统调用可以被调度者用来定义优先级，时间片长度，和其他线程的调度参数，当且仅当调用该系统调用的线程被定义为目标线程的调度者，这个系统调用才能起作用。
L4_Set_Priority	该函数是通用编程接口，从 schedule 系统调用衍生出来，该函数设置指定线程的优先级，前提是此线程是被设置线程的 scheduler 而且新的优先级不能比此 scheduler 的优先级高
L4_MapItem	根据指定的 fpage 和映射位置获取相应的映射项。
L4_unmap	将指定的一个或数个 fpage（位于 MR0, MR1...等消息寄存器中）取消映射。Fpage 是作为 IPC 操作的一部分被映射的。
L4_SystemClock()	获取系统当前时间。

4.2.1 完全调度器

按照 3.2.2.1 设计的思路，我们从内核获得了就绪线程列表和各线程优先级，

从应用程序获得了线程相关调度信息（最坏执行时间和截止时间），就可以在调度器实现具体的调度算法。

在调度器，线程调度相关的数据结构如下所示：

```
struct thread_schedule{
    threadID;   //线程ID
    priority;   //线程优先级
    wcet;       //最坏执行时间
    deadLine;   //截止时间
    ......
};
```

根据 3.2.2.1 节设计的调度算法，当通过参数控制器时，我们根据各线程的优先级，对它们的截止时间进行处理，提高高优先级线程的截止时间，相对降低了低优先级线程的截止时间，在大体公平的原则下，体现出高优先级的优势。

然后对线程按照截止时间长短排成一个队列，用 thread_list 表示，该变量是一个 struct thread_schedule 结构的指针，然后该列表中的线程按顺序通过准入控制器，截止时间短（T 较小）的线程依次通过控制器，直到下一个线程如果通过，$\sum_{i=1}^{n}\frac{C_i}{T_i}>1$，那么，不让该线程通过，这时处理器宽带已满[34]。只对通过准入控制器的线程进行调度，没有通过再外面等待，直到有线程结束，处理器有带宽了，才能进入控制器。

通过了控制器，进入调度队列的线程，我们按照截止时间最短的线程先执行的原则对它们分配处理器。

4.2.2 辅助调度器

按照 3.2.2.2 设计的思路，需要实现一个辅助调度器，我们从内核获得了就绪线程列表和各线程优先级，并且通过一个计时器，每隔一段时间我们从共享内存中得到这段时间内处理器执行过的线程 ID，然后就可以在调度器实现具体的调度

算法。

在辅助调度器，线程调度相关的信息，我们用如下数据结构所示：

```
struct thread_schedule{
        threadID;   //线程ID
        priority;   //线程优先级
        ……
};
```

另外，我们对得到的一段时间内执行过的线程 ID 列表，用 runthread_list 表示，对就绪队列中的线程，它们之中总有一个最大的优先级，我们用 maxpriority 参数表示。

接着，我们用一个循环对这段时间内，没有被处理器执行的就绪线程的优先级进行处理，在我们实现的辅助调度器中，只是对线程优先级加 1，用户可以根据自己的需求实现具体的处理。另外，处理完成后的优先级应该保证不超过最大优先级（因为只要是最大优先级，肯定能够被处理，不会发生饿死的情况），伪代码如下所示：

```
for  thread_schedule.threadID  not  in  runthread_list
     if (thread_schedule.priority < maxpriority )
         thread_schedule.priority ++;
……
```

最后，调度策略完成后，我们调用 L4 微内核提供的 schedule 系统调用，修改线程的调度参数，这里主要是修改各线程的优先级。

4.3 接口改进

在 3.2.3 节，我们设计了新的系统接口，在这一节，我们讲述如何实现这些接口。

4.3.1 在用户层的实现

我们按照第三章中改进的方式，在用户层中，需要先把线程 ID 放入调度器 UTCB 中，然后就可以调用接口了。

修改的系统调用接口 schedule 的定义为：

```
Word ScheduleN ( ThreadId* dest, Word* n, Word TimeControl, Word ProcessorControl,
        Word Prio, Word PreemptionControl, Word* result, Word* old_TimeControl )
```

修改后的通用编程接口实现为：

- Word SetPriority (ThreadId *tid, Word *n, Word Prio, Word *res)
  { return ScheduleN(tid, n, -1, -1, Prio, -1, res) }
- Word SetProcessorNo (ThreadId *tid, Word *n, Word p, Word *res)
  { return ScheduleN(tid, n, -1, p, -1, -1, res) }
- Word SetTimeslice (ThreadId *tid, Word *n, Time ts, Time tq, Word *res)
  { return ScheduleN(tid, n, ts _ 2^{16} + tq, -1, -1, -1, res) }
- Word SetPreemptionDelay (ThreadId *tid, Word *n, Word sensitivePrio,
  Word maxDelay, Word * res) [SetPreemptionDelayN]
  { return ScheduleN(tid, n, -1, -1, -1, sensitivePrio*2^{16} + maxDelay, res) }

4.3.2 在内核中的实现

在内核中，首先获取线程ID列表，以及调度参数列表，一个线程ID对应相同位置的调度参数。获得这些信息以后，内核对批量线程调度参数的修改，实际上就是多次修改单个线程的调度参数，我们不需要大量修改内核，只需要在内核中的schedule函数中，增加一个循环，每次处理一个线程。内核首先根据一个线程ID，查收到该线程的TCB位置，然后根据传进来的调度参数，对它的数据进行修改。完成以后，如果用户要求写入返回结果，我们还需要添加一个循环，把每个线程处理后的结果写入到消息寄存器中。

4.4 性能比较

我们对修改前的schedule系统调用和修改后的系统调用进行测试，比较两者的性能，以下是测试的系统环境。

表 4.2 系统环境

CPU	Celeron 2.66GHz
内存	512M
操作系统	Ubuntu 8.04
编译器	GCC 4.2.4
模拟器	Qemu 0.9.1

通过多次调用该系统调用，修改线程的优先级，得到原系统调用和修改后的系统调用的执行时间。

测试代码如下所示：

```
.....
L4_Clock_t usec1, usec2;
unsigned long totalAPI = 0;
for (i = 1; i <= 6; i++)
{
    usec1 = L4_SystemClock();  //调用系统调用前时间
    totalTime = 0;
    for (int j = 0; j < ROUND; j++)
    {
        L4_Schedule (dest, -1, -1, prio, -1);
    }
    usec2 = L4_SystemClock();  //调用系统调用后时间
    totalAPI = totalAPI + (unsigned long)((usec2 - usec1).raw);
    printf("%d times Total Time: ", (((double)totalAPI)/ ROUND * 10 * i));
    printf("\nAverage API Time: ", ((double)totalAPI)/ ROUND);
.....
}
```

得到原schedule系统调用和改进后系统调用，分别调用多次，所耗费的总时间和平均时间，如表4.3和表4.4所示。

表 4.3 原系统调用测试时间

系统调用次数	总耗费时间（us）	平均时间（us）
10	32.67	3.26
20	72.44	3.62
30	116.56	3.88
40	162.75	4.06
50	185.00	3.70
60	232.45	3.87

表 4.4 新系统调用测试时间

系统调用次数	总耗费时间（us）	平均时间（us）
10	11.41	1.14
20	21.28	1.06
30	29.75	0.99
40	36.54	0.91
50	44.81	0.89
60	52.25	0.87

由上面两个表格可以看出，Schedule系统调用改进后，对于批量修改线程的调度参数，性能得到明显提升。对于一个系统调用，主要耗费时间的地方是进出内核，该改进对于批量修改线程调度参数，可以节省多次进出内核的时间，通过一次进入内核完成原来多次调用原系统调用才能完成的功能。对于改进后的系统调用，一次处理线程越多，平均执行时间越少。

4.5 本章小结

本章依据第三章设计的调度器模型进行实现，详细介绍了用户层调度器与内核、应用线程的通信机制，信息控制机制，和数据记录过程。并依据这个框架，在用户层实现了两个调度器。最后，介绍了如何实现 schedule 系统调用接口的

改进，提高用户层调度器通过这个接口进入内核的性能。

第5章 总结与展望

5.1 本文完成的工作

随着人们对微内核的认识越来越多，微内核技术越来越成熟，微内核系统也得到了越来越广泛的应用，微内核技术成为了近年来一个研究热点，以 L4 为代表的第二代微内核，也得到了越来越多的重视和关注。

本文在研究了微内核的技术现状，以及当前实现方法的不足的基础上，主要完成了三方面的工作：

第一，本文设计和实现了一种基于 L4 微内核上的两层调度模型，在用户态的调度器通过内存映射，IPC 通信，来实现与 L4 内核和应用线程通信，通过记录点控制和数据控制器来获得 L4 内核系统信息，和应用线程调度信息，从而实现自身的调度策略。

第二，本文根据设计的框架，在用户层实现了两个简单的调度器，一个是以 EDF（Earliest Deadline First）算法为基础的调度器，完全取代内核调度器，在用户层实现调度功能。另一个是辅助调度器，对内核的调度算法进行改进利用，主要调度功能的实现还是在内核中完成。

第三，本文改进了一个系统调用接口，该系统调用可以改变线程的调度参数，但只能每次处理一个线程的改动，而在用户层实现的调度器有可能一次修改多个线程的调度信息，那么，就需要多次调用该系统调用，频繁进出内核，损坏系统性能，因此，本文通过修改这个系统调用，使得用户调度器一次进入内核，就可以批量处理多个线程，提高了系统性能。

5.2 未来的展望

本文实现的用户层调度器，对微内核基础上的应用程序实现线程调度，未来，可以在用户层获取内核其他方面的系统资源信息，并实现自身的资源调度策略，来对运行在 L4 微内核上的虚拟机，包括 KVM 虚拟机等，提供资源的调度。这种

方法把内核中所有资源的分配策略都放到用户层实现，内核只需要保留最基本的策略，这样，即符合了微内核小的设计特点，也使得可以实现丰富的资源调度策略，满足虚拟机的要求。特别是对一些运行了3D图形应用的虚拟机[8]，对系统资源的分配策略要求比较高，这种在用户层实现资源调度策略的方法，能满足它们的需求。

参考文献

[1] J.Liedtke. Towards real microkernels. Cornmunications of the ACM, 1996, 39(9): 70~77

[2] 刘晓霞. 操作系统的发展趋势:微内核技术. 西北大学学报自然科学版, 1997, 27(3): 197~200

[3] J. Liedtke. Improving IPC by kernel design. In Proceedings of the 14th ACM Symposium on OS Principles, 1993: 175~188

[4] G. Heiser, K. ElPhinstone, I.Kuz, G.Klein, S.Petters. Towards trustworthy computing systems: taking microkemels to the next level. Operating Systems Review, 2007, 41(3): 3~11.

[5] Hendrik Tews, Hermann Härtig, Michael Hohmuth. VFiasco-Towards a Provably Correct Microkernel. TU Dresden Technical Report TUD-FI01-1, 2001

[6] System Architecture Group. The L4Ka::Pistachio Microkernel. University of Karlsruhe white paper, 2003

[7] L4 microkernel family. http://en.wikipedia.org/wiki/L4_microkernel, 2008

[8] 何家俊, 陈文智. KVM 虚拟机的 3D 图形加速方法研究. 计算机工程, 2010(16)

[9] Microkernel. http://en.wikipedia.org/wiki/Microkernel, 2008

[10] 储泽楠, 李世扬. 浅析操作系统微内核. 新乡学院学报自然科学版, 2008, 25(1): 66~68

[11] Liedtke, Jochen. On u-Kernel Construction. 15th ACM Symposium on Operating Systems, 1995: 237~250

[12] 微内核. http://blog.chinaunix.net/u/14073/showart.php? id=83657, 2006

[13] the mach project. http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html, 1985

[14] 樊建平, 王永杰. MACH:一个新的操作系统核心. 计算机研究与发展, 1992, 29(10): 1~9

[15] 第二代微内核与 L4 微内核. http://blog.chinaunix.net/u/14073/showart.php?id=83658, 2006

[16] Engler, Dl, Kaashock, M.F., O'Toole. Exokernel: an operating system architecture of application-level resource management. 15th ACM Symposium on Operating Systems, 1995

[17] 刘青芬, 程千山. 俄罗斯网络搜索系统. 图书馆杂志, 2001, 20(7): 61~62

[18] Microkernel Construction. http://i30www.ira.uka.de/teaching/courses/lecture.php?courseid=182&lid=en, 2009

[19] The L4Ka Project. http://l4ka.org/, 2009

[20] Ihor Kuz. L4 User Manual API Version X.2. http://ertos.nicta.com.au/publications/ papers/L4UM_x.2.pdf, 2004: 47~61

[21] Kevin Elphinstone. Future directions in the evolution of the L4 microkernel. In Proceedings of the NICTA workshop on OS verification, 2004

[22] Jochen Liedtke. u-kernels must and can be small. 5th Workshop on ObjectOrientation in Operating Systems, 1996

[23] D.Elkaduwe, P.Derrin, K.Elphinstone. Kernel design for isolation and assurance of physical memory. 1st Workshop on Isolation and Integration in Embedded Systems, 2008: 35~40

[24] Björn Döbel. Inter-Process Communication. http://os.inf.tu-dresden.de /Studium/KMB/WS2008/04-Communication.pdf, 2008

[25] L4Ka Team. L4 eXperimental Kernel Reference Manual Version X.2. http://i30www.ira.uka.de/research/documents/l4ka/l4-x2.pdf, 2006: 47~61

[26] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. IEEE Journal of Selected Areas in Communications, 1996, 14(7):1280~1297.

[27] Simon Winwood. Heiser. Flexible scheduling mechanisms in L4. Tech. rep., University of New South Wales, 2000

[28] Jan Stoess, Volkmar Uhlig. Flexible, Low-overhead Event Logging to Support Resource Scheduling. Proceedings of the 12th International Conference on

Parallel and Distributed Systems, 2006: 115~120

[29] Thomas Anderson, Edward Lazowska, and Henry Levy. The performance implications of thread management alternatives for sharedmemory multiprocessors. In Proceedings of the 1989 ACM conference on Measurement and modeling of computer systems, 1989.5: 49~60

[30] Ariel Tamches, Barton P.Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In Proceedings of the fourth USENIX Symposium on Operating systems design and implementation, 1999: 117~130

[31] Karim Yaghmour, Mchel R.Dagenais. Measuring and characterizing system behaviour using kernel-event logging. In Proceedings of the USENIX annual technical conference, 2000: 13~26

[32] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J. R. Goncalves, Oscar Naim, Zhichen Xu, Ling Zheng. Mdl: A language and compiler for dynamic program instrumentation. In Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques, 1997: 201~213

[33] Jan Stoess, Towards effective user-controlled scheduling for microkernel-based systems, ACM SIGOPS Operating Systems Review, 2007: 59~68

[34] Earliest deadline first scheduling. http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling, 2009

[35] Brian D.Marsh, Michael L.Scott, Thomas J. LeBlanc, Evangelos P. Markatos. First-class user-level threads. In Proceedings of the thirteenth ACMSymposium on Operating systems principles, 1991: 110~121

[36] Joshua LeVasseur. Lecture slides: Microkernel construction. http://i30www.ira.uka.de/teaching/, 2006

[37] Sebastian Biemueller, Uwe Dannowski. L4-based real virtual machines-an api proposal. In Proceedings of the First International Workshop on MicroKernels for Embedded Systems, 2007: 36~42

[38] Geoffrey Lee, Charles Gray. L4/darwin: evolving unix. http://www.ertos.nicta.com.au/publications/, 2006

[39] Josep Torrellas, Andrew Tucker, Anoop Gupta. Benefits of cache-affinity

scheduling in shared-memory multiprocessors. In Proceedings of the 1993 ACM conference on Measurement and modeling of computer systems, 1993: 272–274

[40] Bond University, Gold Coast, Queensland. The SawMill Framework for Virtual Memory Diversity. Proceedings of the 6th Australasian Computer Systems Architecture Conference, 2001

[41] 陈超超, 曾庆凯. 采用 SPIN 的 L4 内存管理形式化验证. 计算机工程. 2009, 35(11): 131~133

[42] Using L4Linux-2.6. http://os.inf.tu-dresden.de/L4/LinuxOnL4/use-2.6.shtml, 2007.

[43] Bryan Cantrill, Michael W.Shapiro, Adam H.Leventhal. Dynamic instrumentation of production systems. In Proceedings of the USENIX annual technical conference), 2004: 15~28

攻读硕士学位期间主要的研究成果

- **发表论文**
  - 何家俊，廖鸿裕，陈文智，KVM 虚拟机的 3D 图形加速方法研究，计算机工程（已录用），2010，16 期

- **申请专利**
  - 陈文智，何家俊，姜振宇，杨建华，余奇伟，一种基于微内核的构件化操作系统的评测方法，专利号：200810077584.7。
  - 陈文智，王宽卿，张文博，杨建华，何家俊，一个基于逼近理想排序法的威胁评估与多目标排序方法，专利号：200810077586.6。

- **软件著作权**
  - 何家俊，陈文智，uPcanel 微内核线程及调度子系统，登记号：2009SR020698
  - 何家俊，陈文智，uPcanel 微内核 IPC 管理子系统，登记号：2009SR020686

- **参与项目**
  - 总装预研基金，********，项目编号：********。
  - 国防基础科研重大项目，********，项目编号：********。

致谢

两年多的研究生生涯在不知不觉中就过去了，在这段时间里，我结识了许多良师益友，学到了许多东西，收获良多。

在毕业论文即将完成之际，我首先感谢我的导师陈文智老师，陈老师治学严谨，学识渊博，以及陈老师循循善诱的教导和不拘一格的思路给予我无尽的启迪。陈老师在学习上和生活上给予了我许多帮助，在这里请接受我诚挚的谢意！

感谢我实验室的弟兄们，特别是吴帆同学，黄炜同学，姜振宇同学，王宽卿同学等，从他们身上我学到了许多东西，不仅是技术上的东西，还明白了许多为人处世的道理。同时，是你们和我共同维系着彼此之间兄弟般的感情，使我忘却了远离家乡的孤独。

感谢我的爸爸妈妈，是你们一直支持着我多年来的求学之旅，希望你们永远健康快乐。

同时，感谢所有给予我帮助和关怀的师长、同学和朋友。

何家俊

2010 年于浙大求是园

基于微内核的调度技术研究

作者： 何家俊
学位授予单位： 浙江大学计算机学院